Modelling and Solving Train Scheduling Problems under Capacity Constraints

Shi-Qiang (Samuel) LIU

School of Mathematical Sciences
Faculty of Science
Queensland University of Technology
Australia

Modelling and Solving Train Scheduling Problems under Capacity Constraints

Shi-Qiang (Samuel) LIU
(B.Eng., B.A., M.Sc., M.Eng.)

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy
August 2008

Principal Supervisor: Professor Erhan Kozan
Associate Supervisor: Professor Vo Anh

School of Mathematical Sciences
Faculty of Science
Queensland University of Technology
Brisbane, Queensland 4001, Australia

Statement of Original Authorship

The work contained in this dissertation has not been previously submitted for a degree at any tertiary educational institution. To the best of my knowledge and belief, this dissertation contains no materials previously published or written by another person except where due reference is made.

Signed: _____

Date: _____

Acknowledgements

First, I would like to express my most sincere gratitude to my principal supervisor, Professor Erhan Kozan, for his invaluable advice, support, guidance, patience and kindness throughout the course of this study, which has made the time a very wonderful learning and researching experience in my life. The knowledge and expertise, both within and outside the scope of this research that he has shared with me, have greatly enriched me and prepared me for my future endeavours.

I would also like to express my most sincere gratitude to my associate supervisor, Professor Vo Anh, for his many comments and suggestions, which are very important contributions to this PhD dissertation.

I am extremely grateful to Professor Ong Hoon-Liong, who is my M.Eng. (Master of Engineering) supervisor in the Department of Industrial and Systems Engineering at the National University of Singapore, where I gained fundamental knowledge in scheduling theory and learned computer programming skills.

Additionally, I wish to extend my appreciation to my friends, my colleagues, and all the faculty and staff members of the School of Mathematical Sciences at Queensland University of Technology, for their assistance, encouragements and friendships. Special thanks are given to Wen-Guang Zhao, Tian-Ran (Terry) Lin, Daniel Ngweso, Adam, Bradley, Glen, Cameron, Kenneth, Jennifer, Beth, Paula, Nicole, Zu-Guo Yu, Vicky Chiu, Qian-Qian Yang, Christie Wang, Pastor Ke-Yu Du, Wai-Leong Tang, Feng Zhang, Ya-Ping Wang, Bin Wu, Wan-Zhong Yang and Kai Lv, for all the good times we have shared in Australia.

Finally, I am indebted to my mother, who gave me the support and encouragement I needed to pursue this goal and succeed with my study. I am especially indebted to my beloved wife, Yi-Ping (Michelle) Wang, for her caring when I needed it most.

Glossary

SMS	*Single-Machine Scheduling*
PMS	*Parallel-Machine Scheduling*
FSS	*Flow-Shop Scheduling*
PFSS	*Permutation Flow-Shop Scheduling*
GFSS	*General Flow-Shop Scheduling*
JSS	*Job-Shop Scheduling*
OSS	*Open-Shop Scheduling*
MSS	*Mixed-Shop Scheduling*
GSS	*Group-Shop Scheduling*
DSS	*Dynamic Shop Scheduling*
SMSS	*Static Mixed-Shop Scheduling*
DMSS	*Dynamic Mixed-Shop Scheduling*
BFSS	*Blocking Flow-Shop Scheduling*
BPFSS	*Blocking Permutation Flow-Shop Scheduling*
BGFSS	*Blocking General Flow-Shop Scheduling*
NWFSS	*No-Wait Flow-Shop Scheduling*
LBFSS	*Limited-Buffer Flow-Shop Scheduling*
CBFSS	*Combined-Buffer Flow-Shop Scheduling*
BJSS	*Blocking Job-Shop Scheduling*
PMFSS	*Parallel-Machine Flow-Shop Scheduling*
PMJSS	*Parallel-Machine Job-Shop Scheduling*
BPMFSS	*Blocking Parallel-Machine Flow-Shop Scheduling*
BPMJSS	*Blocking Parallel-Machine Job-Shop Scheduling*
NWPMFSS	*No-Wait Parallel-Machine Flow-Shop Scheduling*
NWPMJSS	*No-Wait Parallel-Machine Job-Shop Scheduling*
NWBPMFSS	*No-Wait Blocking Parallel-Machine Flow-Shop Scheduling*
NWBPMJSS	*No-Wait Blocking Parallel-Machine Job-Shop Scheduling*

Abstract

Many large coal mining operations in Australia rely heavily on the rail network to transport coal from mines to coal terminals at ports for shipment. Over the last few years, due to the fast growing demand, the coal rail network is becoming one of the worst industrial bottlenecks in Australia. As a result, this provides great incentives for pursuing better optimisation and control strategies for the operation of the whole rail transportation system under network and terminal capacity constraints.

This PhD research aims to achieve a significant efficiency improvement in a coal rail network on the basis of the development of standard modelling approaches and generic solution techniques.

Generally, the train scheduling problem can be modelled as a *Blocking Parallel-Machine Job-Shop Scheduling* (BPMJSS) problem. In a BPMJSS model for train scheduling, trains and sections respectively are synonymous with jobs and machines and an operation is regarded as the movement/traversal of a train across a section.

To begin, an improved *shifting bottleneck procedure* algorithm combined with metaheuristics has been developed to efficiently solve the *Parallel-Machine Job-Shop Scheduling* (PMJSS) problems without the blocking conditions.

Due to the lack of buffer space, the real-life train scheduling should consider blocking or hold-while-wait constraints, which means that a track section cannot release and must hold a train until the next section on the routing becomes available. As a consequence, the problem has been considered as BPMJSS with the blocking conditions.

To develop efficient solution techniques for BPMJSS, extensive studies on the non-classical scheduling problems regarding the various buffer conditions (i.e. *blocking*, *no-wait*, *limited-buffer*, *unlimited-buffer* and *combined-buffer*) have been done. In this procedure, an *alternative graph* as an extension of the classical disjunctive

graph is developed and specially designed for the *non-classical* scheduling problems such as the *blocking flow-shop scheduling* (BFSS), *no-wait flow-shop scheduling* (NWFSS), and *blocking job-shop scheduling* (BJSS) problems. By exploring the blocking characteristics based on the alternative graph, a new algorithm called the *topological-sequence algorithm* is developed for solving the non-classical scheduling problems. To indicate the preeminence of the proposed algorithm, we compare it with two known algorithms (i.e. *Recursive Procedure* and *Directed Graph*) in the literature. Moreover, we define a new type of non-classical scheduling problem, called *combined-buffer flow-shop scheduling* (CBFSS), which covers four extreme cases: the *classical* FSS (FSS) with infinite buffer, the *blocking* FSS (BFSS) with no buffer, the *no-wait* FSS (NWFSS) and the limited-buffer FSS (LBFSS). After exploring the structural properties of CBFSS, we propose an innovative constructive algorithm named the *LK* algorithm to construct the feasible CBFSS schedule. Detailed numerical illustrations for the various cases are presented and analysed. By adjusting only the attributes in the data input, the proposed *LK* algorithm is generic and enables the construction of the feasible schedules for many types of non-classical scheduling problems with different buffer constraints.

Inspired by the *shifting bottleneck procedure* algorithm for PMJSS and characteristic analysis based on the alternative graph for non-classical scheduling problems, a new constructive algorithm called the *Feasibility Satisfaction Procedure* (FSP) is proposed to obtain the feasible BPMJSS solution. A real-world train scheduling case is used for illustrating and comparing the PMJSS and BPMJSS models. Some real-life applications including considering the train length, upgrading the track sections, accelerating a tardy train and changing the bottleneck sections are discussed.

Furthermore, the BPMJSS model is generalised to be a *No-Wait Blocking Parallel-Machine Job-Shop Scheduling* (NWBPMJSS) problem for scheduling the trains with priorities, in which prioritised trains such as express passenger trains are considered simultaneously with non-prioritised trains such as freight trains. In this case, no-wait conditions, which are more restrictive constraints than blocking constraints, arise when considering the prioritised trains that should traverse

continuously without any interruption or any unplanned pauses because of the high cost of waiting during travel. In comparison, non-prioritised trains are allowed to enter the next section immediately if possible or to remain in a section until the next section on the routing becomes available. Based on the FSP algorithm, a more generic algorithm called the *SE algorithm* is developed to solve a class of train scheduling problems in terms of different conditions in train scheduling environments. To construct the feasible train schedule, the proposed *SE* algorithm consists of many individual modules including the *feasibility-satisfaction procedure*, *time-determination procedure*, *tune-up procedure* and *conflict-resolve procedure* algorithms. To find a good train schedule, a two-stage hybrid heuristic algorithm called the *SE-BIH* algorithm is developed by combining the constructive heuristic (i.e. the *SE* algorithm) and the local-search heuristic (i.e. the *Best-Insertion-Heuristic* algorithm). To optimise the train schedule, a three-stage algorithm called the *SE-BIH-TS* algorithm is developed by combining the *tabu search* (TS) metaheuristic with the *SE-BIH* algorithm.

Finally, a case study is performed for a complex real-world coal rail network under network and terminal capacity constraints. The computational results validate that the proposed methodology would be very promising because it can be applied as a fundamental tool for modelling and solving many real-world scheduling problems.

Publications Arising from this PhD Research

Liu, S. Q., & Kozan, E. (2008a). Scheduling trains as a blocking parallel-machine job shop scheduling problem. *Computers and Operations Research* (In Press).

Liu, S. Q., & Kozan, E. (2007b). Scheduling a flow shop with combined buffer conditions. *International Journal of Production Economics* (In Press).

Liu, S. Q and Kozan E. (2007c). A Blocking Parallel-Machine Job-Shop-Scheduling Model for the Train Scheduling Problem. *The 8th Asia-Pacific Industrial Engineering and Management Systems Conference*, Kaohsiung, Taiwan, 10.1-10.10.

Liu, S. Q., & Kozan, E. (2007a). A topological-sequence algorithm based on alternative graph for the shop scheduling problems with blocking. *Journal of Intelligent Manufacturing* (Second revision submitted).

Liu, S. Q., & Kozan, E. (2008b). A hybrid shifting bottleneck procedure algorithm combined with metaheuristics for the parallel-machine job-shop scheduling problem. *Journal of Scheduling* (Submitted).

Liu, S. Q., & Kozan, E. (2008c). Scheduling trains with priorities: a no-wait blocking parallel-machine job-shop scheduling model. *Transportation Science* (Submitted).

Contents

List of Figures

List of Tables

Chapter 1 Introduction

CHAPTER OUTLINE

1.1 Background

The railway constitutes an important mode of transportation for both freight and passengers in many countries. The railway industry is a capital intensive industry with large investment in equipment and employees. In addition, operating a railway requires very complex decision-making processes due to the need to schedule several hundred trains over thousands of kilometres distances. Even a small percentage of improvement in the efficiency of the overall operation may bring significant financial return.

Moreover, Australia is the world's largest coal exporting country. Many large coal mining operations in Queensland rely heavily on the rail network to transport coal from various mines to coal terminals at ports for shipment. For example, the coal rail network in central Queensland is shown in Figure 1-1.

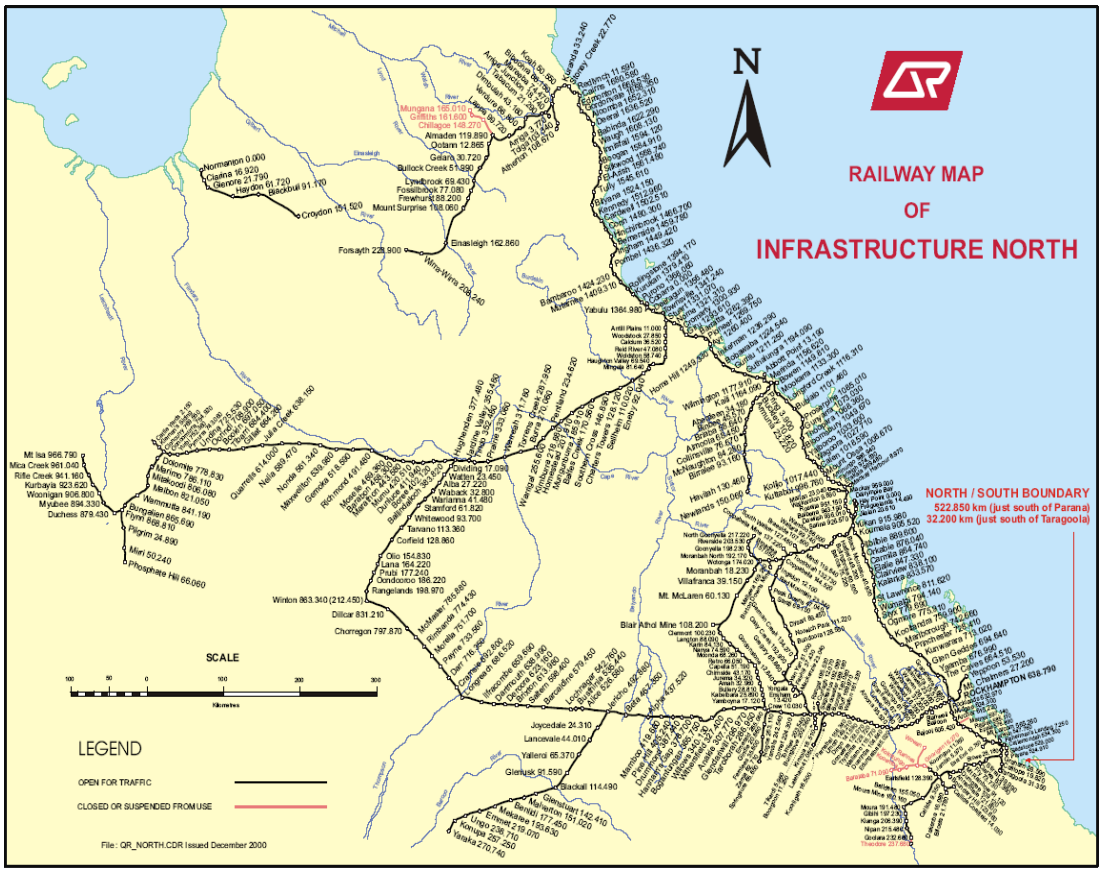

Figure 1-1: Central Queensland coal rail network*

* This figure is provided by Queensland Rail (QR)

Over the last few years, due to the fast growing demand, the coal rail network is becoming one of the worst bottlenecks in the rail industry of Australia. There have been many news articles in *The Australian* newspaper, which addressed the current crisis of coal railing in Australia. To elucidate these facts, some excerpts from these articles are given as below (The Australian, 2007 and 2008):

- **Inefficiency costs coal exports (May 30, 2007)**

  "AUSTRALIA faces the loss of multi-billion-dollar coal export contracts to countries such as Indonesia and Mongolia unless the Queensland Government can clear a tightening bottleneck at its biggest coal port."

  "There is a growing crisis in the far bigger coal sector, with a global steelmaking giant that spends more than $US1billion ($1.22billion) a year on Queensland coal telling *The Australian* yesterday that it was developing other overseas markets because of costly delays and unreliable deliveries by the state-owned railway."

  "Sometimes the ships are waiting more than one month off the (Mackay) coast."

- **Smart State's disgrace (May 30, 2007)**

  "Billions of dollars and thousands of jobs are at stake because Queensland Rail can't get enough coal from the mines to the port to satisfy the demands of international customers."

  "As one coal producer says: We're losing millions of dollars a month on demurrage charges so the crews of these ships can play cards and enjoy the tropical sunshine. How do you think we should explain to our huge customers overseas that we can't meet their orders because our Smart State railway is not delivering the goods?"

  "Frankly, Queensland is the very worst of our suppliers (throughout the world) right now. I have to say that they should be doing everything in every possible way to solve the problem. We need some clear progress. Everybody is losing in this game now."

- **Beattie keeps rail in public hands (May 30, 2007)**

  "Queensland Rail last year made an overall profit of just under $80million."

  "Last year Queensland Rail just carried 162 million tons of coal in Queensland, up from 156 million the year before."

  "Queensland Rail now has overall revenues of $2.5 billion, employs 13,000 people and each day runs about 900 trains."

- **Idle coal wagons infuriate industry in crisis (May 29, 2007)**

"It would take a significant capital upgrade, according to rail chiefs, but many of these older wagons could be used to ease the severe bottleneck at Dalrymple Bay, south of Mackay, where 51 ships were queued yesterday for coal that the railway has been unable to move to the port."

"Queensland Rail responsible for many of the delays, which are costing more than $3 million a day."

"QR acting chief executive Stephen Cantwell said yesterday the proposal to use the idle wagons, including about 200 wagons near Murgon, 200km northwest of Brisbane, was worth examining."

- **$1bn rail logjam hits exports (May 26, 2007)**

"THE bottleneck at one of Australia's biggest coal ports is costing mining companies more than $1 billion a year, threatening hundreds of jobs in the industry and risking the future of exports to key Asian customers."

"As more than 50 ships wait off Queensland's Dalrymple Bay port to load coal, furious coal producers are blaming sheer incompetence by the state-owned railway for the backlog."

"Deputy Premier Anna Bligh said last night she was setting up an urgent process with the Queensland Resources Council and an independent umpire to investigate the capacity problems and find a solution."

- **Jobs go as coal exports choked (May 03, 2007)**

"THE infrastructure crisis on the eastern seaboard has caused the loss of a further 250 jobs, with Coal & Allied blaming bottlenecks at Port Waratah in Newcastle in NSW, the nation's biggest coal export facility."

"It is a really complicated system. We are not getting as much coal through the system as we were led to believe we would."

"The Australian revealed last month that more than 150 ships were anchored off the east coast waiting to be loaded. C&A managing director Douglas Ritchie declined to point the finger at any one aspect of the coal chain."

"This is, frankly, a national disgrace," Mr Ritchie said."

- **$500m system to boost freight rail (June 17, 2008)**

"THE Australian Rail Track Corporation hopes to double capacity on its national freight network by switching to a $500 million computerised management system that allows trains to run safely at shorter intervals."

> "On the traditional rail system you can only have one train between signals, and on the interstate network that could be a distance of 50-60km or more," corporation chief executive David Marchant said."
>
> "You lose all that capacity because you can only have one train between signals."

In this context, the railway industry in Australia demands more new features in the planning and scheduling process and is keen to implement better modelling and solution techniques, in order to improve efficiency and capability of railing coal from mines to ports. The current situation provides great incentives for pursuing better optimisation and control strategies for the operation of the rail transportation system.

By generating better rail schedules, it is possible to increase the utilisation rate of the rail network and reduce the transportation cost. There is one profession that has the necessary skills and expertise to make things efficient and optimal. This profession is *Operations Research* (OR). To produce a better introduction of this PhD research, a brief overview of OR is given in the next section.

1.2 The Overview of OR

The beginning of the activity called *Operations Research* (OR) has generally been attributed to the military services in World War II, as military planners in the UK looked for ways to make better decisions and dealt with allocating scarce resources to satisfy the various military operations at maximum efficiency. The impetus for its origin was the development of radar defence systems for the Royal Air Force, and the first recorded use of the term *Operations Research* is attributed to a British Air Ministry official who constituted a team to do "operational researches" on the communication system at a British radar station. This new approach of picking an "operational" system and conducting "research" on how to make it run more efficiently soon started to expand into other arenas of the war (Hillier and Lieberman, 1995).

The development of OR made its way to the USA a few years after it originated in the UK. Its first appearance in the USA was through the Antisubmarine Warfare Operations Research Group that was led by Phillip Morse, who is widely regarded as the father of OR in the USA. After World War II, OR began to be applied to similar problems in many areas and disciplines including business, economics, logistics, engineering, mathematics, physics, mechanics, artificial intelligence, computer science, etc.

This research is coined as "*operational research*" in the UK and as "*operations* research" in most other English-speaking countries, though OR is a common abbreviation everywhere. However, this name (OR) is somewhat unfortunate, since it is no longer concerned only with operations in some arenas, nor does its application involve any research in a traditional sense. For example, the terms "*operations research*" and "*management science*" are often used synonymously. When a distinction is drawn, management science generally implies a closer relationship to the problems of business management. In addition, OR also closely relates to another term "*industrial engineering*" that takes more of an engineering point of view.

In the years immediately following the end of World War II, OR grew rapidly as many scientists realised that the principles that they had applied to solve problems for the military operations were equally applicable to many problems in the civilian sector. There are at least two factors that played a critical role in the rapid growth of OR. One is the substantial progress in improving the solution techniques available to operations research. After the war, many scientists had participated in OR research or were motivated to pursue research related to OR. Thus, many important state-of-the-art advancements resulted. For example, George Dantzig developed the simplex algorithm for *Linear Programming* (LP) in 1947. Today, LP remains one of the most widely-used OR techniques for many optmisation problems. Once the simplex algorithm had been invented and applied, the development of other techniques followed at a rapid pace. The recent several decades witnessed the development of most of the OR techniques including *nonlinear programming*, *integer programming*, *computer simulation*, *PERT/CPM*, *queuing theory*, *inventory models*, *graph theory*, *game theory*, *scheduling algorithms*, *metaheuristics*,

stochastics, and so on. The scientists who developed these methodologies are inspired from a variety of research areas such as mathematics, statistics, physics, engineering, computer science, business, economics and biology.

Another factor that gave great impetus to the rapid growth of OR is the onslaught of the computer revolution. A large amount of computation effort is usually required to deal with the complex problems typically considered by OR researchers. The rapid development of computer science (hardware and software) causes a tremendous boom to OR, as computer can perform arithmetic calculations thousands or even millions of times faster than a human being can.

Typical OR problems range from short-term problems such as scheduling and inventory control, to long-term problems such as strategic planning and resource allocation. OR is concerned with optimal decision making in, modelling of, and coordinating for the complex systems that originate from real life. The methodology used in OR for optmisation problems is often characterised as the "science of making the future better".

A typical OR project comprises three basic steps: (1) building the model; (2) solving the problem; and (3) implementing the result. Thus, the progress of OR can be represented in three dimensions: expanding the range of problems, increasing efficiency and effectiveness of solution techniques, and applying the obtained results in practice.

In a sense, the procedure for solving the train scheduling problem is a typical process of scientific inquiry in these three aspects. Dirneberger (2002) wrote an article about the role of OR (or Industrial Engineering or Management Science) for improving the service of the railroad industry. He indicated that train schedules were usually designed to meet the needs of the railroad, not the needs of the customers. Transportation is a service; however, even today, the railroad industry does not have a good benchmark for measuring customer satisfaction. Despite the fast development of railroad technology and an increase in the demand for premium-priced, time-definite service, the railroad industry grew up slowly and had a tendency to remain static until threat of competition from the motor-carrier

industry. It is apparent that the railroad industry needs to improve its efficiency. This raises the question, "how does railroad industry improve its efficiency?" There is one profession that has the necessary skills to optimise systems, make things efficient, and maximise profits. This profession is OR.

One way to improve the productivity of the railroad industry involves improving the infrastructure. However, this requires a lot of money to expand capacity by improving existing facilities and costs far more to build a new network. The railroad industry has to figure out the best way to utilise their existing resources. Many different things including locomotive utilisation, yard capacity, crew availability, track structure, and traffic levels all play a role and combine to determine railing efficiency. One more important point indicates that OR will not be able to realise its full potential for optimisation until railroad industries run on a set of optimal (or near-optimal) train schedules and adhere to these schedules. To yield optimal train schedules will require OR techniques to work.

1.3 The Scope of this Dissertation

As there are a great many questions that the railroad industry faces in regards to improving railing efficiency, OR will certainly have a profound positive impact on the future development of the rail industry.

Broadly speaking, one way to improve productivity by OR for the railway industry is to determine the best way to utilise its existing facilities. This type of problem is called ***Train Planning*** that includes locomotive dispatching, wagon coupling, yard management, crew availability, and track maintenance. Another area where OR can be applied is ***Train Scheduling*** which includes train timetable (train schedule) optimisation, conflict resolution and capacity usage maximisation. This PhD study has mainly fallen into the second category, because the aim of this research is to achieve a significant efficiency improvement in the coal rail network on the basis of the development of modelling approaches, algorithms analysis, and solution techniques.

In this PhD dissertation, the train scheduling problems for the coal rail network are generally modelled as a *No-Wait Blocking Parallel-Machine Job-Shop-Scheduling* (**NWBPMJSS**) problem or its classified cases with different conditions in train scheduling environments. To the best of our knowledge, no researchers in the literature address these new scheduling problems. Moreover, many innovative algorithms are proposed to solve these new problems. The proposed algorithms have been implemented and validated using real-life data from Queensland Rail and a consultation company. The outcomes show that the proposed methodology would be very promising as a fundamental implementation tool for the real-life train scheduling problems.

The below diagram is drawn for showing the logic structure of this PhD research.

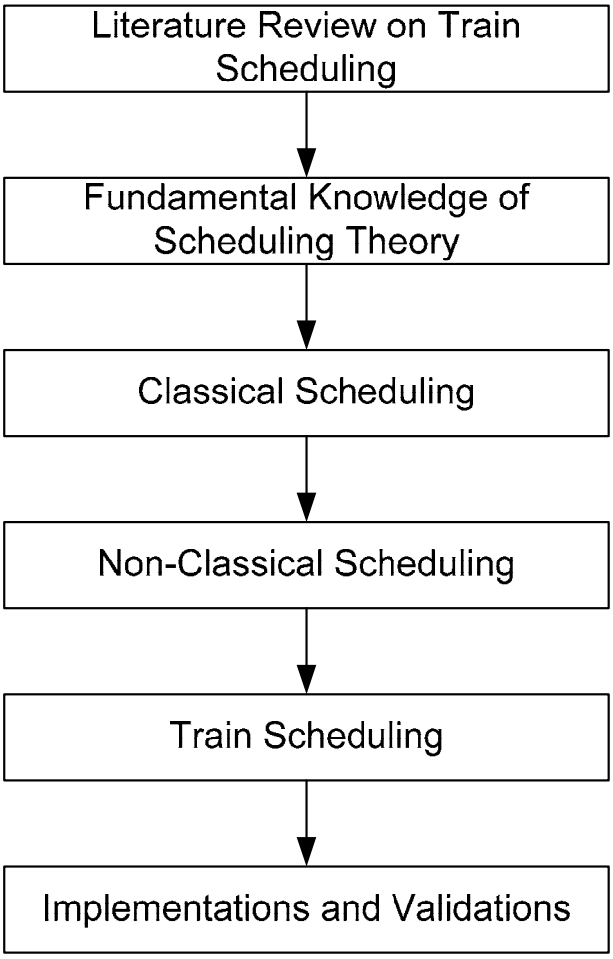

Figure 1-2: The diagram for showing the logic structure of this PhD research

The content of this dissertation is organised as follows. In the next chapter, some previous research works about train scheduling are summarised. An overview of

scheduling theory is given in Chapter 3. Chapter 4 presents a general description for the ***classical*** scheduling problems that consist of single-stage systems (i.e. the basic single-machine scheduling problem, the extensions of the basic single-machine scheduling problem and the parallel-machine scheduling problem); and multi-stage systems (i.e. the flow-shop, job-shop, open-shop, mixed-shop and group-shop scheduling problems). Chapter 5 deals with the ***non-classical*** scheduling problems by considering the *blocking* (*no-buffer*) conditions, *no-wait* conditions, and the limited capacity of inter-machine buffer storage. With the help of scheduling theory on *classical* and *non-classical* scheduling, the train scheduling problems are modelled as the PMJSS, BPMJSS and NWBPMJSS problems in Chapter 6. In Chapters 5 and 6, by exploiting the structural properties that are analysed by mathematical programming, disjunctive graph and alterative graph, many innovative algorithms are proposed for solving the following scheduling problems listed below:

1. *blocking flow-shop-scheduling* (BFSS);
2. *no-wait flow-shop-scheduling* (NWFSS);
3. *limited-buffer flow-shop-scheduling* (LBFSS);
4. *combined-buffer flow-shop-scheduling* (CBFSS);
5. *blocking job-shop-scheduling* (BJSS);
6. *parallel-machine job-shop-scheduling* (PMJSS)
7. *blocking parallel-machine flow-shop-scheduling* (BPMFSS);
8. *no-wait parallel-machine flow-shop-scheduling* (NWPMFSS);
9. *blocking parallel-machine job-shop-scheduling* (BPMJSS);
10. *no-wait parallel-machine job-shop-scheduling* (NWPMJSS)
11. *no-wait blocking parallel-machine flow-shop-scheduling* (NWBPMFSS);
12. *no-wait blocking parallel-machine job-shop-scheduling* (NWBPMJSS);

among which some new problems like CBFSS, BPMJSS and NWBPMJSS are originated in this research.

To implement and validate the proposed methodology for train scheduling, a case study is conducted in Chapter 7 to obtain the optimal (or near-optimal) train timetables in a complex real-world coal rail network under network and terminal

capacity constraints. Finally, the main contributions of this PhD research and the directions of future research are summarised in Chapter 8.

Chapter 2 Literature Review on Train Scheduling

CHAPTER OUTLINE

The literature review on train scheduling has been finalised in the following three areas, namely: *capacity analysis*; *exact solution techniques*; and *heuristic solution techniques*. The details of some previous research on train scheduling are summarised in the following sections.

2.1 Capacity Analysis

In the 1970s, the compilation of long-term timetables for high-density rail service with multiple classes of trains on the same track was done by humans, not computers.

Petersen (1974) developed an analytic model of the mean running time for trains on a single track railway. In this model, trains operating at several different speeds in each direction are permitted. Priority systems are included in the model to control train behaviours when meets and overtakes occur and delay times due to implementing these priority systems are formulated. In addition, it is assumed that the departing times for trains are independent random variables that are uniformly distributed; thus the resulting mean running times and delays are found by solving a set of linear equations.

Fukumori (1980) proposed an algorithm that uses the range-construction search technique to schedule the timing and pass-through relations of trains. His program can determine how the timing of certain trains constraints the timing of others, find possible time regions and pass-through relations, and evaluate the efficiency of train movement for each pass-through relation.

Welty (1991) questioned whether precision train scheduling might be an entirely Utopian vision. From an operating standpoint, precision train scheduling brings great benefits: 1) train meets and passes would be made without having to stop or slow down appreciably; 2) there would be positive separation between trains; 3) classification yards would have a steady flow of traffic, without periods when there is not much to do or periods when trains have to be held out on the main line because there is no place to accommodate them; 4) motive power would be available and ready when needed; 5) trains would not be delayed for lack of rested

crews. In all, with the development of railroad technology, greater precision in the rail freight service is possible. However, the question still remains: What are the best ways to get it (i.e. precision train scheduling) done?

Higgins, Ferreira and Kozan (1995a) present analytically-based models to quantify the amount of delay risk associated with each track sections and the train schedule as a whole. Three main types of delays are modelled, namely: terminal/station delays; track related delays; and rolling stock related delays. The models can be used to prioritise investment projects designed to improve timetable reliability. The effect of timetable changes on likely reliability can also be modelled. Using the risk models developed, it is possible to assess the likely effect of removing/adding sidings for passing and crossing purposes, under single line train operations. The effect of changing the timetables and the assumptions regarding delays, due to terminal congestion, track related problems, and rolling stock, are summarised.

Higgins, Ferreira and Kozan (1995b) described the development and use of a model designed to optimise train schedules on single line rail corridors, when the priority of each train in a conflict depends on an estimate of remaining crossing and overtaking delay. This priority is used in a branch and bound procedure to allow the determination of optimal solution quickly. The model was developed with two major applications in mind, namely: as a decision support tool for train dispatchers to schedule trains in real time in an optimal way; and as a planning tool to evaluate the impact of timetable changes as well as railway infrastructure changes.

Investment decisions on upgrading the number and location of these sidings can have a significant impact on both customer service and rail profitability. **Higgins, Kozan and Ferreira** (1996a and 1997a) put forward a model to determine the optimal positions of a set of sidings (crossing loops) on a single-track rail corridor. The sidings are positioned to minimise the total delay and train operating costs of a given cyclic train schedule, with the allowance of non-constant train velocities and non-uniform departure times. A new approach is proposed to solve this problem, whereby the positions of the sidings are determined simultaneously with the optimal schedule using a decomposition procedure. In this way, the final solution will

produce improved siding positions as well as the corresponding optimal resolved train schedule. Importantly from an operational perspective, the measures in objective function include train travel time and train arrival reliability. In addition, simulations are used to demonstrate how the model can be used to determine the required number of sidings given a pre-defined level of service.

Caprara, Monaci, and Toth (2001) proposed an approach to solve the train timetabling problem (TTP) for a set of trains that does not violate track capacities and satisfies some operational constraints. They concentrated on the problem of a single one-way track linking two major stations with a number of intermediate stations in between, taking into account several additional constraints that arise in real-world applications. These constraints include 1) automatic and manual signals in the track segments between two consecutive stations; 2) station capacities; 3) a prescribed timetable for a subset of the trains; 4) maintenance operations; 5) periodic trains; and 6) two parallel alternative lines connecting two major stations. The computational results were presented on real-world instances from the Italian railway company.

Dorfman and Medanic (2004) developed a local feedback-based travel advance strategy (TAS) to schedule trains in a railway network, using a discrete event model (DEM) of train advances along lines of the railway. Until now, for most research on train scheduling, there have been two drawbacks: 1) despite the tremendous speed of today's supercomputers, an integer programming problem with a constraint for every train and siding in a railway network still takes an unreasonable long time to solve; and 2) if such a solution is implemented, whenever a single train is unable to keep to the schedule, the entire problem must be recomputed from the current state of the network. This proposed TAS can quickly handle perturbations in the schedule and is shown to perform well on three time-performance criteria while maintaining the local nature of the strategy. If the local strategy leads to a deadlock, a capacity check algorithm is applied to prevent deadlock but requires additional non-local information. The proposed TAS is also extended for the network with the double-track sections and variable train priorities.

Abdekhodaee et al. (2004) investigated the integration of scheduling a rail network with some of the operations in a coal terminal system with limited capacity, because the rail network and terminal systems are tightly-coupled and experience a high service demand for the expensive infrastructure that makes efficient operations essential. They proposed mixed integer programming models for these two systems and then discussed the merits and disadvantages of devising such an approach and the significance of promoting coordination between the operational functions of these two systems.

Kozan and Burdett (2005) proposed approaches on the determination of railway capacity and discussed the significance of some factors on capacity. An accurate model is developed to calculate railway capacity considering previously unaddressed aspects. Capacity and pricing are two key issues for organizations involved with open track access regimes. A train access charging methodology is therefore developed and incorporated into the railway capacity determination model.

Burdett and Kozan (2006) developed capacity analysis techniques and methodologies for estimating the absolute traffic carrying ability for a railway system under a wide range of defined operational conditions, which include the proportional mix of trains, the directions, the length of trains, the planned dwelling times of trains, the presence of crossing loops and intermediate signals in corridors. The proposed techniques are illustrated in a case study, which demonstrated that capacity analysis approaches can be used to quantify the potential interaction effects that may occur in interrelated corridors.

2.2 Exact Solution Techniques

The public railway sector in many parts of the world has increased the awareness of the need for a quality service that must be offered to its customers.

Ingolotti et al. (2005) developed a software system, named the Decision Support System (DSS), for solving and plotting the single-track railway scheduling problem

(STRSP) quickly and efficiently. The STRSP problem is formulated as a constraint satisfaction problem (CSP), which is solved by using different stages to translate problems into mathematical models by means of mixed integer programming tools. The DSS allows the users (Railway companies) to interactively specify the parameters of the STRSP problem and guarantees that constraints are satisfied and the optimized timetable are obtained.

Epstein et al. (2005) developed a mathematical programming model to determine the optimal dispatching times for complex rail networks in densely populated metropolitan areas, in which some portions of the rail network may consist of single-track lines while other locations may consist of double-track or triple-track lines. The model is solved by a branch-and-bound algorithm with the help of transferring trackage to a general network graph and applying propagation rules. They also demonstrated the efficiency of the proposed branch-and-bound algorithm by comparing it to CPLEX, a commercially available integer program solver, on an actual rail network in Los Angeles County.

Salido et al. (2005) indicated that many railway scheduling problems can be modelled as constraint optimisation problems (COP). In this model, the railway scheduling was considered as a problem subject to a number of constraints that describes railway infrastructure, train service requirements and reasonable time-intervals for waiting and transits. Collaborating with the National Network of Spanish Railways (RENFE), they developed a topological constraint optimization technique for solving periodic train scheduling. The topological optimization technique divides the proposed model into subproblems such that a traffic pattern is generated for each subproblem. Then, these traffic patterns are periodically repeated to compose the entire running map. The computational experiments show that this technique can improve the results obtained by well-known COP solvers, such as LINGO and CPLEX software.

Linder and Zimmermann (2005) considered minimizing the operational cost of train schedules, which depend on choosing different train types of diverse speed and cost. A mixed integer programming model was proposed for modelling this train scheduling problem. Although it seems to be impossible to directly solve the model

of practical sizes within a reasonable amount of time, suitable decomposition can be applied to achieve good performance. In the first part of the decomposition, only the train type related constraints stay active. In the second part, the remaining constraints are satisfied using a relaxation technique. This decomposition idea provides a cornerstone for an algorithm integrating cutting plane and branch-and-bound to optimize the railway networks in Germany and the Netherlands.

Lindner (2004) developed a fundamental mathematical model named the *Periodic Event Scheduling Problem* (PESP), which finds a feasible schedule for some periodically recurring events subject to certain constraints. There are many criteria for evaluating schedules. Linder extended it by an objective function representing the operational costs of a schedule. The resulting model is called *Minimum Cost Scheduling Problem* (MCSP). With the help of the mixed integer programming (MIP) formulation and polyhedral methods like pre-processing techniques, valid inequalities, a specific relaxation, lower bound determination, a branch-and-bound, and a cutting plane procedure, Linder solved real-world instances of the PESP and the MCSP within a reasonable amount of time.

Zhou and Zhong (2004) dealt with a double-track train scheduling problem with multiple objectives. Focusing on a high-speed passenger rail line in an existing network, the problem is to minimize both: 1) the expected waiting times for high-speed trains (efficiency criterion); and 2) the total travel times of high-speed and medium-speed trains (effectiveness criterion). By applying two practical priority rules to model acceleration and deceleration times, the problem is decomposed and formulated as a multi-mode flow-shop scheduling problem. A branch-and-bound algorithm with an effective dominance rule is developed to generate Pareto solutions for the bicriteria scheduling problem, and a beam search algorithm with utility evaluation rules is used to construct non-dominated solutions. The authors illustrated the methodology and evaluated the performances of the proposed algorithm by a case study based on Beijing-Shanghai high-speed railway in China.

Ghoseiri, Szidarovszky and Asgharpour (2004) developed a multi-objective optimization model for the passenger train scheduling problem on a railway network, which includes single and multiple tracks, as well as multiple platforms

with different train capacities. In their study, lowering the fuel consumption cost is the measure of satisfaction of the railway company and shortening the total passenger-time is regarded as the passenger satisfaction criterion. The solution of the problem consists of two steps: 1) Pareto frontier is generated using the ε-*constraint* method; and 2) based on the obtained Pareto frontier detailed multi-objective optimisation is performed using the distance-based method with three different types of distances.

Sahin, Ahuja and Cunha (2004) developed an integer programming formulation for the train dispatching problem based on a space-time network. The train dispatching problem aims to determine detailed timetables over a rail network in order to minimize deviations from the planned schedule, and is concerned with helping dispatchers make the right decisions about which train should stop in order to avoid conflicts as updated data about train positions become available. The model also includes some realistic constraints that have not been previously considered such as maximum allowable delays. Heuristic algorithms (i.e. IP-Based Heuristic, Simulation-Based Construction Heuristic and Greedy Enumeration Heuristic) are proposed to solve the model.

2.3 Heuristic Solution Techniques

Many researchers have dealt with train scheduling problems by heuristic algorithms. In recent years, the following papers in the literature have addressed this issue.

Tsen (1995) addressed the following two major issues: 1) modelling the railway line planning problem as a modified job-shop scheduling problem; 2) demonstrating the benefits of using multiple representations in A-Teams. The goal is to benefit from the many existing algorithms developed for the job-shop scheduling problems, and to combine them into an A-Team to solve the line planning problem. One of the major hurdles to achieve this goal was to find a good deadlock-avoidance mechanism, because deadlock situations do not occur in job-shops, but they may occur quite frequently on railway lines. In his research, Tsen developed a deadlock

avoidance mechanism and integrated it into many job-shop scheduling algorithms. Then, Tsen combined the various algorithms, many of them using different representations, into a multiple representation A-team to solve the line planning problem.

Cai and Goh (1994) proposed an algorithm that is based on a local optimality criterion in the event of a potential crossing conflict. In addition, the problem can be generalised to cater for the possibility of overtaking when trains travel at different speed. In practice, such a problem is often required to be solved in real time. Hence, a quick heuristic that allows a good feasible solution to be obtained in a predetermined and finite number of steps or in polynomial time is most desired. Simulation results for two non-trivial cases are presented to demonstrate the efficiency and effectiveness of the proposed algorithm.

Higgins, Kozan and Ferreira (1997b) took great initiative in applying metaheuristic techniques to solve the single-track railway scheduling problem which is known to be NP-Hard with respect to the number of conflicts. The heuristics applied include a local-search heuristic, a genetic algorithm, a tabu search and two hybrid algorithms with an improved neighbourhood structure. Comparisons made between each of the heuristics with and without constraints on computation time show that the genetic hybrid algorithms were within five percent of the optimal solution for at least ninety percent of the test problems.

Oliveira and Smith (2000) modelled the single-track railway scheduling problem as a special case of the job-shop scheduling problem. It was achieved by considering the train trips as jobs, which will be scheduled on track sections regarded as machines. A train trip may have many tasks (job operations) that consist of traversing from one point to another on a track. The objective of this model is to minimize the total delay when conflicts can be resolved. A conflict occurs when two trains occupy the same track section at the same time. In this paper, conflicts are resolved by applying the shortest processing time (SPT) rule to reschedule the tasks on all track sections on which a conflict is found. In addition, some practical constraints are incorporated into this model. The computation

experiments are carried out using the dataset of 19 Higgins's problems provided from Higgins (1996b).

Caprara, Fischetti and Toth (2002) proposed a single-track train timetable model. In this model, each train connects two given stations along the track and may have to stop for a minimum time in some of the intermediate stations; and trains can overtake each other only in correspondence to an intermediate station with a specified minimum time interval. A graph theoretic formulation is proposed for solving the problem by a direct multi-graph in which nodes correspond to departure/arrivals at a certain station at a given time in time. The problem is formulated as an integer programming model that can be relaxed in a Lagrangian way. A novel feature of this model is that the variables in the relaxed constraints are only associated with nodes of the graph. Embedded within a heuristic algorithm, this feature allows a considerable speed-up in the solution procedure. The authors reported extensive computational results on real-world instances provided from the Italian railway company.

Chew et al. (2001) developed a computerized train-operator scheduling system based on an optimization approach, which has been implemented at Singapore Mass Rapid Transit (SMRT). The optimization approach involves a bipartite matching algorithm for the generation of night duties and a tabu search algorithm for the generation of day duties. The system can automate the train-operator scheduling process at SMRT, produce favourable schedules in comparison with the manual process, and handle the multiple objectives inherent in the crew scheduling system.

Pacciarelli and Pranzo (2001) presented a tabu search algorithm for a railway scheduling problem by means of the alternative graph. The alternative graph is an extension of the disjunctive graph of Roy and Sussman (1964). Based on the alternative graph formulation for the railway scheduling problem, the idea of a tabu search algorithm is described.

Furthermore, **Pacciarelli** (2002) developed a similar alternative graph model to solve the complex factory scheduling problems. This is because in the disjunctive graph models, it is assumed that a job can be stored for an unlimited amount of time

and intermediate buffers have infinite capacities. However, for many real-life situations such as train scheduling or factory scheduling, the buffer capacity has to be taken into account. By applying the alternative graph, the model proposed can effectively analyse and solve a number of complex practical scheduling problems for which there have were no successful methodologies. The performance of a new heuristic both on real data and several instances from the literature is discussed in relation to the job-shop scheduling problem.

2.4 Summary

It has been observed that most mathematical programming models in previous research are specific and only suitable for some special scenarios, and the solution techniques had to resort to taking advantage of structural properties of the proposed models. On the other hand, for the above research results, it is very hard to identify the borders between generic and special, feasible and infeasible, hard and easy, theoretical and practical. Thus, we wonder whether there is a better way to deal with train scheduling problems, and intend to explore whether there is a more standard, generic and convenient methodology to analyse, model, and solve train scheduling problems.

As discussed in Chapter 1, OR can be considered as a process of inquiry; that is, as a procedure for answering questions, solving problems and developing more effective procedures for answering questions and solving problems. Some researchers argue that common-sense inquiry is qualitatively oriented, whereas scientific inquiry is quantitatively oriented. It should be realised that much of the common knowledge is itself based on the products of yesterday's science.

In this sense, a basic way to solve the real optimisation problem is to use a "toolbox" of standard well-solved OR problems. This is because the real problems can be easily analysed, modelled (decomposed, aggregated or simplified) and solved by the right choice of standard tools from the toolbox. This process classifies the real problem into many standard types and pinpoints the difficulty as hard or easy.

In this PhD study, a very useful "toolbox" that will be adopted for working out the train scheduling problems is *scheduling theory*. Thus, before dealing with train scheduling problems, an overview on scheduling theory is given in the next chapter.

Chapter 3 Scheduling Theory

CHAPTER OUTLINE

3.1 Scheduling Function

Scheduling is the allocation of resources over time to perform a collection of tasks (Baker 1974). This general definition of the term conveys two different meanings. First, scheduling is a body of theory: a collection of principles, models, techniques and logical conclusions. Thus, much of what we learn about scheduling can apply to other theories and therefore has general *conceptual* value. Second, scheduling is a decision-making function: a process of determining a desirable schedule. In this sense, much of scheduling theory can be applied to other kinds of decision making and therefore has general *practical* value.

Practical scheduling problems arise in a variety of real-life situations. In most cases, however, scheduling does not become a concern until some fundamental planning questions are answered: e.g. "*is the availability of resources known*?", and "*is it appropriate to consider it as a scheduling problem*?". For instance, the delivery of perfect health care may require the designation of medical services, facility allocation, equipment utilisation, and personnel deployment. Once these answers are determined, it is then possible to deal with hospital scheduling. The scheduler then takes information as given and determines how to allocate the available resources to perform the specified tasks. When a tentative schedule is constructed, the scheduler can evaluate it and convey his evaluation to the user. The user may not be satisfied with the performance achieved by the tentative schedule and may alter the planned resource capacities (or even tasks), thereby providing revised input for the scheduler. The interplay between these two roles might be repeated in this manner over several exchanges before a final decision is reached.

With this background in mind, scheduling decisions are reached as a systematic procedure, which is comprised of at least four primary stages: 1) formulation; 2) analysis; 3) synthesis; 4) evaluation (Baker 1974).

Formulation is the first stage that is often critical and subtle, because good decisions are seldom expected without a clear definition of tasks and criteria. For

many combinatorial optimisation problems, mathematical programming models like *integer programming* (IP) are very helpful in the formulation stage.

Analysis is the detailed procedure of examining the elements of a problem and their relationships. In practice, there are limits on the capacity of the available resources and technological restrictions on the order in which tasks can be performed. This stage is aimed at specifying the decision variables, the relationships among them, and the constraints that they must obey.

Synthesis is the process of building the feasible solution to the problem. A solution is any feasible resolution of these constraints, so that solving a scheduling problem mounts to answering two kinds of questions: 1) Which resources should be allocated to perform each task as required? 2) By what time should each task be efficiently performed? The processes of synthesis are aided by a familiarity with suitable algebraic, logical, simulation or graphical models. For example, one of the simplest and most widely used models is the *Gantt chart*, which is a graphical representation of resource allocation over time. Analysis of graphical relationships can yield inferences about the characteristics of a given schedule, while manipulation of the graphical elements can provide comparative information about alternative schedules. The following chapters will examine other graphical models such as *disjunctive graph* and *alternative graph* models. These models contain the important attributes and relationships that indicate how feasible solutions can systematically be constructed. The process of building a feasible solution is often a vital part for many complicated scheduling problems. A caveat needs to be added regarding the role of models in scheduling function. Indeed, when a model is actually a faithful representation of reality, it can become an integral part of the scheduling function. Even coarse and somewhat oversimplified models can also be of value, for their fundamental role is to represent the general behaviour and essential properties of related scheduling problems.

Finally, *evaluation* is the process of comparing these feasible alternatives and selecting the most desirable one in terms of the given criteria. The design of an efficient evaluation process is definitely a sophisticated art. For example, one can devise for the same problem various kinds of *Tabu Search* metaheuristic algorithms

which are essentially different with respect to the basic elements that consist of the initial solution, move, neighbourhood structure, aspiration function, tabu list, perturbation, searching strategy and stopping rules.

In a sense, it is the scheduling theory that has summarised and systematised these methodologies providing a useful "toolbox" to perform the scheduling function for many real-world scheduling projects.

3.2 Scheduling Terminology

Scheduling has been the subject of extensive research since the early 1950s. The main focus is on the efficient allocation of resources to activities over a given amount of time. Many of the early developments in the field of scheduling were motivated by problems arising in manufacturing. It was natural to employ the vocabulary of manufacturing when describing scheduling problems. Now even though scheduling work is of considerable significance in many non-manufacturing areas, the terminology of manufacturing is still frequently used. Thus, resources are usually called "*machines*" and basic task modules are called "*jobs*". Sometimes, jobs may consist of several elementary tasks referred to as "*operations*".

In addition, the range of application areas for scheduling theory goes beyond manufacturing to include agriculture, hospital, transport, computer, etc. Therefore, it is possible to encounter, for example, a problem in the scheduling of outpatient visits to specialists in a diagnostic clinic and to find the system described generically as the processing of "jobs" by "machines" (Baker, 1974).

3.3 Scheduling Complexity

A major theme in recent research has been the use of complexity theory to classify scheduling problems as polynomially solvable or NP-hard. The NP-hardness of a problem suggests that it is unlikely to find an optimal solution without the use of an

essentially enumerative algorithm, for which computation times will increase exponentially with problem size. To obtain the exact solutions of NP-hard scheduling problems, branch-and-bound, dynamic programming or integer programming is usually applied. In most cases, these exact algorithms have been successful in solving problems of reasonable size. There are, however, some classes of problems that have resisted attempts to design a satisfactory solution procedure, implying that exact algorithms may be unable to solve problems with more than a handful of jobs and the solutions generated by simple methods may be far from the optimum. Recently, such problems can be efficiently tackled by local search methods with metaheuristic mechanisms such as *Simulated Annealing* (SA), *Threshold Accepting* (TA), *Tabu Search* (TS) and *Genetic Algorithm* (GA).

If the problem is NP-hard, finding an exact solution is apt to be costly or impractical. In practice, it may be acceptable to use a metaheuristic to find an approximate solution for a NP-hard problem. There is clearly a trade-off between the computational time spent and the quality of the solution found. The performance of metaheuristics is often evaluated empirically by analysing how far the solution deviates from a lower bound or upper bound (the best known solution value) of the problem.

3.4 Scheduling Classification

The basic scheduling problems that we consider in this thesis can be described as follows. There are m machines, which are used to process n jobs. A schedule specifies, for each machine i and each job j, one time interval throughout which processing is performed on job j by machine i. A schedule is feasible if each job can only be processed on one machine and each machine can only process one job at a time, and also if it satisfies various requirements imposed on the jobs and machines relating to the specific problem type. The machine environment, the job characteristics, optimality criteria, and inter-machine buffer conditions distinctly specify the scheduling model and the problem type.

Different configurations of machine environment are possible. In most classical cases, however, we restrict our attention to deterministic and nonpreemptive scheduling. In addition, it is assumed that the capacity of inter-machine buffer storage is infinite and all machines are available to process jobs at time zero. According to the machine environment and job characteristics, classical machine scheduling problems can be divided into two production systems: *single-stage system* and *multi-stage system*.

A single-stage system requires only one operation for each job, whereas in a multi-stage system there are jobs that require several operations processed on different machines.

Single-Machine Scheduling and Parallel-Machine Scheduling

Single-stage systems involve either a single machine referred to as the single-machine scheduling (SMS) problem, or m machines in parallel referred to as the parallel-machine scheduling (PMS) problem.

In multi-stage systems, there are three well-known types of basic shop scheduling problems in the literature: the *flow-shop scheduling* (FSS) problem, the *job-shop scheduling* (JSS) problem and the *open-shop scheduling* (OSS) problem. All such systems that we consider contain n jobs, each of which comprises m operations processed on m machines, but having a different function. Another two types of multi-stage systems called the *mixed-shop scheduling* (MSS) problem and *group-shop scheduling* (GSS) problem have recently been introduced but receive little attention.

Flow-Shop Scheduling

In a flow shop, each job is processed on machines 1, …, m in this unidirectional fixed order. Moreover, if the processing order of the jobs on each machine is the same for every machine, the flow-shop scheduling (FSS) problem is called the *permutation flow-shop scheduling* (PFSS) problem, which is a special case of the *general flow-shop scheduling* (GFSS) problem. In most cases, without specific

emphasis, the FSS problem is default as the permutation case to be solved. The reason for this will be given in the next Chapter.

Job-Shop Scheduling

In a job shop, each job has a prescribed routing through the machines (i.e. *machine sequence*), but this machine sequence may be different for distinct jobs.

Open-shop scheduling

In an open shop, even the machine sequence of each job is unrestricted and forms a part of the decision process.

Mixed-Shop Scheduling

In practice, multi-stage systems may be a mixture of the above three "pure" shops, called the *mixed-shop scheduling* (MSS) problem. In a mixed shop, according to job characteristics, some jobs may be flow-shop-type jobs which have unidirectional fixed machine sequences; some jobs may be job-shop-type jobs which are defined by different fixed machine sequences; and some jobs may be open-shop-type jobs for which machine sequences of these jobs are undetermined. In other words, in a mixed shop, the machine routes of jobs can either be fixed or unrestricted. In this sense, the MSS problem is a more generic case of the above three "pure" shop scheduling problems.

Group-Shop Scheduling

Another more generic multi-stage scheduling problem is called the *group-shop scheduling* (GSS) problem. The group-shop scheduling problem is described as follows: Given a set of operations O that is partitioned into subsets of $J = \{J_1, ..., J_n\}$, and subsets $M = \{M_1, ..., M_m\}$, where n is the number of jobs, m is the number of machines. J_i is the set of operations which belong to job i. M_k is the set of operations which have to be processed on machine k. For each job i, the operations are also partitioned into g groups $G = \{G_1, \ldots, G_g\}$. The operations in the same group are unrestricted while operations in two distinct groups satisfy the precedence relationship between these two groups. If every group consists of only one

operation, this GSS problem turns out to be a JSS problem. If for all J_i, the number of groups is equal to 1 (i.e. g=1), the GSS problem is equivalent to an OSS problem.

Job characteristics are the second element to specify the scheduling problem types. A job is characterized by its availability for processing, any setup requirements on the machine, any dependence on other jobs, and whether interruptions in the processing of its operations are allowed. The additional assumptions of job characterization (e.g. non-simultaneous arrival, dependent jobs, and sequence-dependent setup times) will result in the extensions of the basic model.

The optimality criterion or objective function is the third element to specify the problem types. For each job j, the processing time t_j, ready time r_j, due date d_j and positive weight w_j are known in advance to serve as input information. Given a schedule, we can compute for job j: the completion time (C_j); the flowtime ($F_j = C_j - r_j$); the lateness ($L_j = C_j - d_j$); the tardiness ($T_j = \max\{C_j - d_j, 0\}$) and the unit penalty $\delta_j = 1$ if $C_j > d_j$, $\delta_j = 0$ otherwise. Thus, some commonly used optimality criteria may involve the minimisation of:

- $C_{max} = max_j\ C_j$, the maximum completion time (Makespan).
- $L_{max} = max_j\ L_j$, the maximum lateness.
- $\Sigma_j\ (w_j)C_j$, the total weighted completion time.
- $\Sigma_j\ (w_j)T_j$, the total (weighted) tardiness.
- $\Sigma_j\ \delta_j$, the number of tardy jobs.
- $(\Sigma_j\ F_j)/n$, the mean flowtime.
- $F_{max} = max_j\ F_j$, the maximum flowtime.

In recent years, a considerable amount of interest has arisen in *non-classical* shop scheduling problems for considering the limited inter-machine storage buffer, such as *blocking*, *no-wait*, *limited-capacity* buffer conditions. The interest appears to be motivated as much by real-life applications as by theoretic inquiry.

No-Wait Shop Scheduling

An important class of shop scheduling problems is characterised by a no-wait production environment which typically arises from characteristics of the

processing technology itself, or from the absence of storage capacity between operations of a job. In a no-wait shop, each job must be processed continuously from its start on the first machine, to its completion on the last machine, without any interruption on machines and without any waiting in between the machines. For instance, scheduling problems in no-wait shops often arise in petrochemical production environments and hot metal rolling industries, where chemicals or metals have to be processed at continuously high temperatures. Another example of a no-wait situation arises in scheduling prioritised trains (e.g. express passenger trains) because passenger trains should traverse continuously from the first station to the last station without any interruption.

Blocking Shop Scheduling

Scheduling problems with blocking conditions arise in many real-world manufacturing processes where no intermediate buffer storage is available. In such situations, a job that has completed processing on a machine may remain there until a downstream machine becomes available, but this prevents another job from being processed there.

Limited-Buffer Shop Scheduling

On the boundary of the no-wait/blocking scheduling environment are a number of interesting issues. In real life, for example, a flow shop with limited inter-machine buffer storage is a very common manufacturing environment. Much can be learned about this problem by studying the blocking (no-buffer) environment or unlimited-buffer environment, which is an extreme case of the finite-buffer capacity problem. Actually, problems of this type create a relatively new direction of research.

Parallel-Machine Shop Scheduling

The parallel-machine shop scheduling problem is another generalisation of the classical shop scheduling problem. In a classical shop, it is assumed that the number of same-type machines for processing one operation is only one. In practice, however, parallel-machine shop scheduling problems often arise in the area of flexible manufacturing systems in which one operation may be processed on parallel machines. In other words, the number of same-type machines for

processing an operation may be more than one. A typical example arises in train scheduling problems.

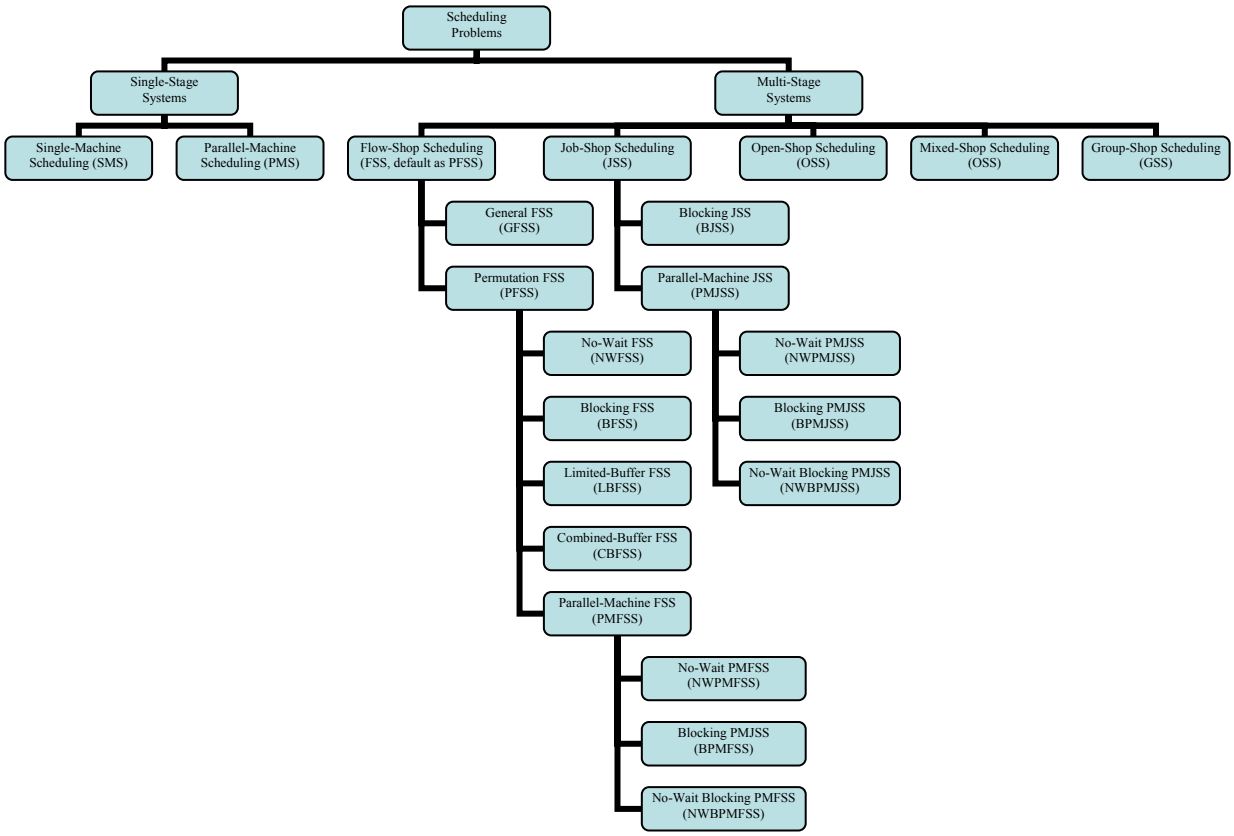

Figure 3-1: The classification of scheduling problems studied in this dissertation

As above, according to different definitions of machine environment, job characteristics and inter-machine buffer conditions, the classification of scheduling problems studied in this dissertation can be illustrated by a tree structure exhibited in Figure 3-1. At least 22 types of scheduling problems have been studied in this thesis.

3.5 Scheduling Representation

It is convenient to use representation notations to denote a large family of scheduling problems. In this thesis, we propose a four-field descriptor $\alpha|\beta|\theta|\gamma$ to indicate problem type, where α represents the machine environment, β defines

the job characteristics, θ regards the inter-machine buffer conditions, and γ is the optimality criterion.

The first field α denotes the type of the machine environment (for example, *F*: a flow shop, *J*: a job shop, *O*: an open shop, *M*: a mixed shop, *G*: a group shop). If the problem is with a fixed number of *m* machines, the above descriptor is followed by a parameter *m*. Additionally, a superscript P is used to denote the *parallel-machine* shop scheduling problems. For example, J^P represents the machine environment of the *Parallel-Machine Job-Shop Scheduling* (PMJSS) problem.

The second field β specifies the job characteristics. If the number of jobs is fixed, then the number of jobs *n* is included in the second field β. If the job characteristic β includes the parameter *pmtn*, then operation preemption is allowed; otherwise preemption is forbidden as default. In other words, for non-preemption cases, it is processed to completion without interruption once processing begins on a job.

The fourth field γ specifies the optimality criterion, which involves the minimisation of $\gamma \in \{C_{max}, L_{max}, \Sigma_j (w_j)C_j, \Sigma_j (w_j)T_j, \Sigma_j \delta_j, (\Sigma_j F_j)/n, max_j F_j\}$.

Usually, the third field θ is ignored as default for the *classical* scheduling problem with the unlimited inter-machine buffer storages. For instance, the problem of constructing a schedule with minimum makespan for fifteen nonpreemptive jobs and ten machines in a classical mixed shop (the mixture of the open-shop-type, flow-shop-type, and job-shop-type jobs) can be denoted by

$$M10|n_O = 5, n_F = 5, n_J = 5|C_{max},$$

where n_O, n_F, and n_J denote the number of the open-shop-type jobs, the number of the flow-shop-type jobs, and the number of the job-shop-type jobs, respectively.

To represent the buffer conditions between each pair of two successive machines, the third field θ is particularly added to denote the *non-classical* scheduling problem with limited capacity of inter-machine buffer storage. For example, a 5-machine 10-job *blocking flow-shop scheduling* (BFSS) problem with the objective of minimising makespan can be denoted as:

$$F5|10\ |b_{1,2}=0,\ldots,\ b_{m\text{-}1,m}=0|C_{max}$$

where $b_{j,j+1}=0$ means that there is no buffer storage between two consecutive machines M_j and M_{j+1}. More details will be given in Chapter 5 for introducing a new non-classical scheduling problem that combines various buffer conditions in a flow shop environment.

3.6 Summary

In this chapter, we briefly discussed some of the fundamental knowledge about scheduling theory, including scheduling function, scheduling terminology, scheduling complexity, scheduling classification and scheduling representation.

In general, numerous types of scheduling problems to be studied in this thesis are classified according to machine environment, job characteristics, optimality criterion and inter-machine buffer conditions. To particularly specify the inter-machine buffer conditions, we adopt a four-field representation scheme to distinguish the problem type.

Chapter 4 Classical Scheduling

CHAPTER OUTLINE

FOR PUBLISHED PAPERS CONTRIBUTE TO CHAPTER 4

Liu, S. Q., & Ong, H. L. (2002). A comparative study of algorithms for the flowshop scheduling problem. *Asia-Pacific Journal of Operational Research, 19*, 205-222.

Liu, S. Q., & Ong, H. L. (2004). Metaheuristics for the mixed shop scheduling problem. *Asia-Pacific Journal of Operational Research, 21*(4), 97-115.

Liu, S. Q., Ong, H. L., & Ng, K. M. (2005a). Metaheuristics for minimizing the makespan of the dynamic shop scheduling problem. *Advances in Engineering Software, 36*, 199-205.

Liu, S. Q., Ong, H. L., & Ng, K. M. (2005b). A fast tabu search algorithm for the group shop scheduling problem. *Advances in Engineering Software, 36*, 533-539.

4.1 Introduction

As discussed in Chapter 3, according to the characteristics of the relationship between the jobs and the operations, the scheduling problems have been divided into two main categories: ***single-stage systems*** and ***multi-stage systems***.

Furthermore, the multi-stage scheduling problems are distinguished as ***classical*** or ***non-classical***, in terms of the capacity of inter-machine buffer storage. If the capacity of buffer storage between any two successive machines is *infinite*, this problem is regarded as the *classical* type problem. Otherwise, it is referred to as one of *non-classical* type problems that will be dealt with in the next chapter. Note that all of the scheduling problems studied in Chapter 4 are classical.

The structure of this chapter is organised as follows. Section 4.2 gives the preliminaries of the *single-machine scheduling* (SMS) problem, and then presents the ***basic*** SMS models and its extensions, followed by discussing the *parallel-machine scheduling* (PMS) problems. Section 4.3 provides the detailed description of multi-stage systems including the classical static *flow-shop scheduling* (FSS), *job-shop scheduling* (JSS), *open-shop scheduling* (OSS), *mixed-shop scheduling* (MSS) and *group-shop scheduling* (GSS) as well as *dynamic shop scheduling* (DSS) problems. In Section 4.4, a generic state-of-the-art methodology is presented for solving all of these five static shop scheduling problems and dynamic shop scheduling problems. The concluding remarks are given in the last section.

4.2 Single-Stage Systems

In single-stage production systems, each job consists of only one operation. Single-stage production systems involve either a single-machine or m machines in parallel. Single-stage production systems studied in this dissertation consist of the *basic single-machine scheduling* (SMS) problem, the extensions of the SMS problem, and the *parallel-machine scheduling* (PMS) problem.

4.2.1 Preliminaries of Single-Stage Systems

The ***basic*** *single-machine scheduling* (SMS) model is characterised by the following five conditions (Baker 1974):

1. A set of *n* independent, single-operation jobs is available for processing at time zero.
2. Setup times for the jobs are independent of job sequence and can be included in the processing times.
3. Job information is known deterministically.
4. Each machine is continuously available and is never kept idle while work is waiting.
5. Once processing begins on a job, it is processed to completion without interruption.

Under these conditions, the total number of distinct solutions to the single-machine problem is therefore *n*!, which is the number of different permutations of *n* elements. If a schedule can be characterised by a permutation of integers, it is called a ***permutation schedule***. This concept is very useful for understanding the difference between the *permutation* FSS (PFSS) problem and the *general* FSS (GFSS) problem to be discussed in the next section.

In dealing with job attributes for the basic SMS problem, it is useful to distinguish between information that is known in advance and information that is achieved as the result of scheduling decisions. Three basic pieces of information that help to describe jobs in the SMS model are:

- Processing time (p_j): the amount of processing time required by job *j*.
- Ready time (r_j): the point at which job *j* is available for processing.
- Due date (d_j): the point at which the processing of job *j* is due to be completed.

The assumption in Condition 1 means that r_j is equal to 0 for all jobs. Under Condition 2, the processing time p_j will generally include both direct processing time and setup time.

Information that is achieved as a result of scheduling decisions represents the output from the scheduling function, and it is usually convenient to use capital letter to denote this type of data:

- Completion time (C_j): the point at which the processing of job *j* is finished. Note that quantitative measures for evaluating single-machine schedules are usually functions of job completion times.

- Flowtime (F_j): the amount of time job *j* spends in the system: $F_j = C_j - r_j$. The flowtime represents the interval between a job's departure and arrival.

- Lateness (L_j): the amount of time by which the completion time of job *j* exceeds its due date: $L_j = C_j - d_j$. It is important to note that lateness represents poorer service than requested. In some situations, distinct penalty and other costs will be associated with positive lateness, but no benefits will be associated with earliness (i.e. a negative L_j value).

- Tardiness (T_j): the amount of time when job *j* fails to meet its due date, or zero otherwise. That is, $T_j = \max\{0, | L_j |\}$.

Thus, some performance measures are defined in the following:

- Mean flowtime: $\overline{F} = \frac{1}{n}\sum_{j=1}^{n} F_j$
- Mean tardiness: $\overline{T} = \frac{1}{n}\sum_{j=1}^{n} T_j$
- Maximum flowtime: $F_{\max} = \max_{1 \le j \le n}\{F_j\}$
- Maximum tardiness: $T_{\max} = \max_{1 \le j \le n}\{T_j\}$
- Number of tardy jobs: $N_T = \sum_{j=1}^{n} \delta(T_j)$ $\begin{cases} \delta(T_j) = 0, \ if \ \ T_j = 0 \\ \delta(T_j) = 1, \ if \ \ T_j > 0 \end{cases}$

Actually, if a performance measure Z is a function of the set of job completion times C_j, i.e. $Z = f(C_1, C_2, \ldots, C_n)$, we can say that the measure Z is ***regular*** if the scheduling objective function is to minimise Z and Z can increase only if at least one of the completion times in the schedule increases.

The measures introduced above are all regular measures. Since most scheduling problems will deal only with regular measures, this definition is significant because it is usually desirable to restrict attention to a limited set of schedules called a ***dominant set***. For example, for the basic SMS problem without machine idle time and without job splitting (preemption), there is an optimal schedule in a dominant set with respect to any regular measure.

4.2.2 Basic Single-Machine Scheduling

It is convenient to use brackets to indicate the positions of a SMS sequence. For example, $J_{[5]} = J_2$ means that the fifth job in sequence is Job 2. In this fashion, $p_{[1]}$ refers to the processing time of the first job in sequence.

The rule of sequencing the jobs in a non-decreasing order of processing times ($p_{[1]} \le p_{[2]} \le \ldots \le p_{[n]}$) is known as *shortest processing time* (SPT). For the basic SMS problem with the objective of minimising mean flowtime, the mean flowtime can be minimised by SPT rule. In addition, for the basic SMS problem with the objective of minimising mean lateness, the optimization is also achieved by SPT rule.

Another sequencing rule is called *earliest due date* (EDD) with the non-decreasing order of due dates ($d_{[1]} \le d_{[2]} \le \ldots \le d_{[n]}$). For the basic SMS problem with the objective of minimising the maximum lateness or tardiness, the optimisation can be accomplished by EDD rule.

In a sense, the basic SMS problem is fundamental in scheduling theory. Although it is rather simple with relatively small resource allocation requirement and

dimensions, the set of feasible solution space is still quite large and the determination of an optimum for some criteria (e.g. minimising the mean tardiness) may become a formidable combinatorial problem.

4.2.3 Extensions of Basic Single-Machine Scheduling

The development of more general models and methods extends the applicability of sequencing theory by relaxing some of the assumptions in the basic model (Baker, 1974 and Rinnooy, 1976).

In the extensions of the basic single-machine scheduling problem, there are several ways to generalise Condition 1 given in Section 4.2.1 (i.e. assuming all jobs are simultaneously available for processing at time zero), such as allowing different ready times or dependent sets of jobs. For example, an immediate consequence of allowing different ready times is the requirement to re-examine the new questions if allowing inserted idle time on the machine (relaxing Condition 4) and job preemption (relaxing Condition 5). To illustrate the role of these two factors, consider a two-job SMS example with different ready times and the objective of minimising total tardiness, shown in Table 4-1.

Table 4-1: A two-job SMS example with different ready times

Job j	1	2
r_j	0	1
p_j	5	2
d_j	7	2

As shown in Figure 4-1, the sequence (1, 2) satisfies conditions 4 and 5 by avoiding inserted idle time and preemption. This sequence has a total tardiness of 5.

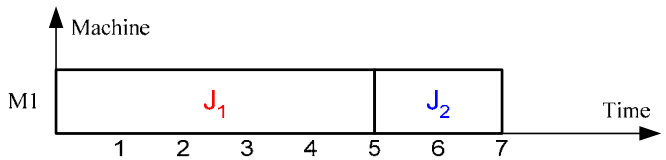

Figure 4-1: Schedule of jobs without inserted idle time in a two-job SMS example

When inserted idle time is permitted, the sequence shown in Figure 4-2 leads to a total tardiness of 2. That is, a better schedule that minimises the total tardiness can be obtained.

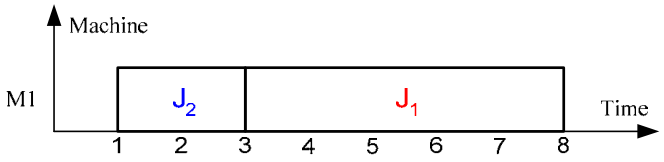

Figure 4-2: Schedule of jobs with inserted idle time allowed in a two-job SMS example

Furthermore, if a job can be preempted and later continued from the point at which the interruption occurred, then a total tardiness of 1 can be achieved, as illustrated in Figure 4-3.

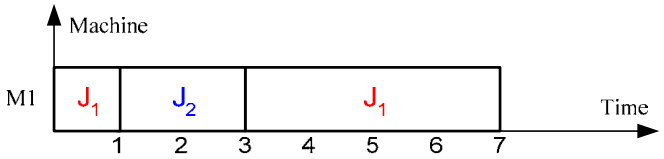

Figure 4-3: Schedule of jobs with inserted idle time and preemption allowed in a two-job SMS example

The case in which a job is resumed from the point at which it is interrupted, as shown in Figure 4-3, is called the *preemption* mode. When preemption prevails, some sequencing rules are essentially unchanged to obtain the optimal SMS schedule with a certain criterion. For example, for the problem of minimising the maximum tardiness when preemption is allowed, the optimal sequencing rule is still EDD.

Condition 2 can also be generalised by allowing sequence-dependent setup times. In this situation, explicit modifications must be made. Obviously, it is not valid to absorb the setup time s_{ij} for job j into the processing time p_j, when job i precedes job j. The time interval in which job j occupies the single machine is expressed as $s_{ij} + p_j$, where s_{ij} is the setup time required for job j after job i is completed, and p_j is the amount of processing time required to complete job j. In the *basic* SMS problem, the time to complete all jobs (i.e. makespan) is a constant. With sequence-dependent setup times, however, the makespan depends on which

sequence is chosen. Such an extensional SMS model can be treated as a *travelling salesman problem* (TSP). The proof is explained below.

The makespan ($C_{\max}$) of one given SMS schedule is computed by:

$$C_{\max} = \sum_{j=1}^{n+1} s_{[j-1],[j]} + \sum_{j=1}^{n} p_j \tag{4.1}$$

where states 0 and $n+1$ are treated as the idle states, i.e. $s_{[0],[1]} = 0$ and $s_{[n],[n+1]} = 0$. It is evident that the problem of minimising the makespan is equivalent to minimising the first summation in Eq. (4.1), as the second summation is a constant. This summation represents the total non-productive times in the full sequence, beginning and ending in the idle state. This type of structure represented by the SMS problem with the sequence-dependent setup times and the objective of minimising the makespan is modelled as a TSP. In the TSP, the salesman wishes to choose a tour that will take him to each city once and that will return him to the origin city. Given the distance between all pairs of cities, the salesman's task is to find a tour with the minimum total travel distance. In the sequencing problem, the setup time s_{ij} corresponds to the distance between city *i* and city *j*.

The generalisation of Condition 3 may involve the use of probabilistic or stochastic methods. Therefore, the basic SMS problem is changed from a *deterministic* model to a *stochastic* model. For example, suppose that job *j* has a random processing time and also a random due date, with the *expected value* $\mathrm{E}[p_j]$ or $\mathrm{E}[d_j]$. Thus, the objective is changed to minimise the maximum expected lateness, $\max_j\{\mathrm{E}[L_j]\}$, and the optimal sequence rule is *expected earliest due date* (EEDD):

$$\mathrm{E}[d_{[1]}] \le \mathrm{E}[d_{[2]}] \le \ldots\ldots \le \mathrm{E}[d_{[n]}].$$

To sum up, the extensions considered in this section can greatly enrich the basic scheduling problems and demonstrate the fundamental role of the basic SMS model in scheduling theory.

4.2.4 Parallel-Machine Scheduling

Similar to the basic SMS model, a *basic* parallel-machine scheduling (PMS) model supposes that there are *n* single-operation jobs simultaneously available at time zero, and that there are *m* identical machines available for processing, and that a job can be processed by at most one machine at a time.

In the basic SMS model, the makespan remains unchanged for any sequence. Therefore, there is no makespan problem to be considered in the basic SMS case. In the PMS case, the makespan problem is no longer trivial.

For the PMS model with the objective of minimising the makespan, McNaughton (1959) presented an elementary method when the jobs are independent but job preemption is permitted. With preemption allowed, the processing of a job may be interrupted and remaining processing can be completed subsequently, perhaps on a different machine. In this case, the minimum makespan or lower bound, M^*, is simply calculated by:

$$C^*_{\max} = \max\{\frac{1}{m}\sum_{j=1}^{n} p_j, \max_j[p_j]\} \tag{4.2}$$

Then, an optimal PMS schedule can be easily constructed by using Eq. (4.2). It is worth noting that this problem does not usually have a unique solution and the construction method produces one of many potential optimal schedules. In addition, the method makes no attempt to minimise the number of preemptions.

If job preemption is prohibited, the problem of minimising the makespan is somewhat more difficult. A simple yet effective procedure for constructing a schedule involves the use of the *longest processing time* (LPT) rule.

When the job set is dependent, the PMS problem with the objective of minimising the makespan is considerably more difficult. The algorithm for solving it consists of two procedures, the labelling phase and the scheduling phase (Hu 1961 and Muntz-Coffman 1969). In addition, we will illustrate in detail how to extend the Jackson rule (1955) to solve an extensional PMS problem with different ready times and delivery times in Chapter 5. .

For more references, please refer to Mokotoff (2001) who presented a detailed overview of the fruitful research on the PMS problems.

4.3 Multi-Stage Systems

Compared with single-machine and parallel-machine scheduling problems, in the multi-stage systems, each job generally consists of *m* operations, each of which requires a different machine.

Within the new setting of multi-stage systems, the conditions that characterise the multi-stage systems are similar to the conditions of the basic SMS model. Minimising the makespan is the main criterion in most studies for multi-stage systems. In the literature, there are well-known types of multi-stage systems, that is, the *flow-shop scheduling* (**FSS**), *job-shop scheduling* (**JSS**) and *open-shop scheduling* (**OSS**) problems. In comparison, very few researchers addressed another two types of multi-stage systems, i.e. the *mixed-shop scheduling* (**MSS**) and *group-shop scheduling* (**GSS**) problems.

4.3.1 Flow-Shop Scheduling

In many manufacturing factories, a number of operations have to be processed in the same order (i.e. the same machine sequence) for every job. The machines are assumed to be set up in series and the environment is referred to as a flow shop. The *flow-shop scheduling* (**FSS**) problem is stated as follows. In a flow shop, there are *n* jobs that have to be performed on *m* machines. Each job consists of *m* operations, each of which must be processed on machines 1,2,…,*m* in this unidirectional order. The objective is to minimise the maximum completion time, i.e. makespan.

When searching for the optimal schedule of a FSS instance, the question always arise whether it suffices merely to determine a *permutation* of *n* integers, which

implies that in a permutation FSS schedule, the same job sequence occurs on every machine.

Johnson's Problem

The *two-machine* flow-shop scheduling (FSS) problem with the objective of minimising the makespan is also known as *Johnson's problem*, of which the solution is a *permutation* schedule. The results originally obtained by Johnson (1954) are now standard fundamentals in scheduling theory. In the formulation of this problem, job j is characterised by processing time p_{j1} required on machine 1, and p_{j2} required on machine 2 after the operation on machine 1 is completed. An optimal sequence can be determined by *Johnson's rule* for ordering pairs of jobs.

***Johnson's Rule**:*

Job i proceeds job j in an optimal sequence if $\min\{t_{i1},\ t_{j2}\} \le \min\{t_{i2},\ t_{j1}\}$

In practice, an optimal sequence is easily constructed with an adaptation of this rule. At each stage, a job should fill either the first or last available position. Johnson's Algorithm is given as below.

Johnson's Algorithm

Step 1: Find $\min_i\{t_{i1},\ t_{i2}\}$.

Step 2a: If the minimum processing time occurs on machine 1, place the associated job in the *first* available position in sequence. Go to *Step 3*.

Step 2b: If the minimum processing time occurs on machine 2, place the associated job in the *last* available position in sequence. Go to *Step 3*.

Step 3: Remove the assigned job from consideration and return to Step 1 until all positions in sequence are filled.

For the *three-machine* FSS problem with the makespan criterion, it is also sufficient to consider only permutation schedules in the search for an optimum. In terms of applying the above Johnson's algorithm, these results require only that instead of

seeking the minimum processing time, the first step is to seek a minimum in the following form: $\min_i \{t_{i1} + t_{i2}, t_{i2} + t_{i3}\}$.

Dominant Permutation Schedules

For the general (non-permutation) FSS problem, there are $n!$ different job sequences possible on each machine, and therefore $(n!)^m$ different schedules to be examined. It may not be sufficient to consider only permutation schedules in which the same job sequence occurs on each machine. On the other hand, it is not always necessary to consider $(n!)^m$ alternatives in determining an optimal FSS schedule, because the following two dominance properties indicate how much of a reduction is possible.

Property 1: With respect to any regular measure of performance, it is sufficient to consider only schedules in which the same job sequence occurs on machines 1 and 2.

Property 2: With respect to the makespan criterion, it is sufficient to consider only schedules in which the same job sequence occurs on machine m-1 and m.

The implication of these two dominance properties is that in searching for an FSS optimal schedule, it is necessary to consider different job sequences on different machines with these two general exceptions. That is,

a) For any regular measure, it is sufficient for the same job sequence to occur on machines 1 and 2, so that $(n!)^{m-1}$ schedules constitute a dominant set.
b) When $m > 2$, it is sufficient for the same job order to occur on machines 1 and 2, as well as machines $m-1$ and m, so that $(n!)^{m-2}$ schedules constitute a dominant set.

Therefore, for the general (non-permutation) FSS problem, a direct application is to observe that it is sufficient to consider only permutation schedules in the search for an optimum for the following cases:

1) Regular performance measure, and $m = 2$.
2) Makespan criterion, and $m = 2$ or $m = 3$.

Permutation Flow-Shop Scheduling

As the same job sequence occurs on every machine, the solution of the *permutation flow-shop scheduling* (PFSS) problem is simply represented by a permutation of the job indices 1, 2, 3, …, n.

Although permutation schedules are not guaranteed to provide optimal solutions for the *general flow-shop scheduling* (GFSS) problem when $m \geq 4$, most researchers follow a tradition in the literature and still concentrate on the PFSS problem, i.e. finding the best *permutation* schedule. Therefore, without special emphasis, the FSS problem is considered as the PFSS problem as default.

Solution Techniques for Permutation Flow-Shop Scheduling

Many researchers proposed a number of solution procedures for the *permutation flow-shop scheduling* (PFSS) problem because it is a fundamental multi-stage system. These include exact and approximate solution techniques for finding an optimal or near-optimal sequence. The exact techniques (such as branch and bound, dynamic programming and integer programming) solve the problem in principle, but the computation time and the memory required to keep track of calculations is prohibitive even for small-sized problems. Constructive heuristic algorithms, though they do not necessarily provide the optimal solution to the problem, are an efficient and economical way of getting a good solution. We can find the description of these simple heuristics such as the algorithms proposed by Palmer (1965), the algorithms proposed by Gupta (1970 and 1971), the CDS algorithm proposed by Campbell *et al*. (1970) and the NEH algorithm proposed by **N**awaz, **E**nscore and **H**am (1983). Among them, the NEH algorithm is recognised as the best polynomial heuristic in practice. The NEH algorithm is presented as follows:

Step 1: Order the n jobs by decreasing sums of processing times on all the machines.

Step 2: Take the first two jobs and schedule them in order to minimise the partial makespan as if there were only two jobs.

Step 3: For $k \leftarrow 3$ to $k \leftarrow n$ do:

Insert the k^{th} job at each k possible position, which minimises the partial makespan.

In addition, various metaheuristics were proposed for the PFSS problem since the 1990s. Two *simulated annealing* (SA) algorithms were proposed by Osman and Potts (1989) and by Ogbu and Smith (1990). Both approaches compared *shift* (*insert*) and *swap* neighbourhood moves and demonstrated that the shift neighbourhood performed better in extensive computational experiments. Ogbu and Smith attributed this to the comparatively large size of neighbourhood, whereas Osman and Potts suggest it may depend on the objective function. An evaluation of these two approaches by Ogbu and Smith found that they yield similar results, although the algorithm of Osman and Potts is marginally more effective.

Afterwards, several *tabu search* (TS) metaheuristics for the PFSS problem with makespan criterion were proposed by Widmer and Hertz (1989), Taillard (1990), Reeves (1993), Nowicki and Smutnicki (1996) and Grabowski and Wodecki (2004).

Widmer and Hertz (1989) used the swap neighborhood and adopted a tabu list that prevents the interchanged jobs returning to its previous position. Taillard (1990) suggested an improvement to each of the key components in the method of Widmer and Hertz. In his algorithm, the shift neighborhood was used and two tabu lists were adopted. One tabu list contains the values of the makespan, and the other stores the forbidden positions for jobs. Taillard also derived a method for evaluating all shift neighbors in $O(mn^2)$ time. Based on the computational tests, he claimed that the shift neighbourhood is better than the swap neighbourhood proposed by Widmer and Hertz.

Reeves (1993) proposed the use of a restricted version of the shift neighbourhood in which a specific subset of jobs is eligible for insertion at each iteration. After all subsets are considered, new subsets are created by random partitioning.

Nowicki and Smutnick's tabu search algorithm (1996) used a restricted version of the shift neighbourhood, where the block structure of some critical path determines the eligible neighbours. Using ideas from the analysis of Taillard, together with the

observation that the block structure limits the number of positions to which a job is allowed, Nowicki & Smutnicki established that all neighbours of their algorithm can be evaluated in $O(m^2n)$ time.

Grabowski and Wodecki (2004) developed a very efficient tabu search quite similar to Nowicki and Smutnick's method, but employing new properties associated with the blocks on the critical path. The presented properties and ideas can be applied in any local search procedure.

Reeves (1995) also proposed a *genetic algorithm* (GA) that uses the reorder crossover. In his computational results with the same computation time limit for each algorithm, the proposed genetic algorithm is comparable with the simulated annealing of Osman and Potts.

Several of the above metaheuristic algorithms have been tested on the benchmarks generated by Taillard (1993), thus enabling some additional comparisons to be made. From the various sets of computational results, it was recognised in the literature that the tabu search algorithm of Nowicki and Smutnicki (1996) was the current champion for the PFSS problem with makespan criterion.

Some conclusions can be drawn from the studies on neighbourhood search with metaheuristics for the PFSS problem. First, shift (insert) neighbourhood is the best neighbourhood structure when minimising the makespan. Second, it is often advantageous to restrict the search to an eligible subset of the neighbourhood moves. Third, efficient estimation instead of exact computation for evaluating a neighbouring solution has a substantial effect on reducing the computation time of the proposed metaheuristics.

Comparative Study on PFSS and GFSS

If the job sequences on different machines may be various in finding a FSS solution, this type of FSS problem is referred to as the *general* FSS (GFSS) problem. In the PFSS problem, there are $n!$ different job sequences possible because the job

sequence is the same for every machine, and therefore there are $(n!)^m$ different schedules to be examined for the GFSS problem.

In the literature, it is not always necessary to consider $(n!)^m$ schedules in determining an optimum. It is sufficient to consider only permutation schedules for the two-machine and three-machine GFSS problems with makespan criterion. Most researchers henceforth concentrated on the PFSS problem, i.e., finding the best permutation schedule for the GFSS problem. However, for $m > 3$, permutation schedules are not guaranteed to provide optimal solutions to the GFSS problem.

In order to figure out whether it is sufficient to only consider the permutation schedules for the GFSS problem when $m > 3$, Liu and Ong (2002) did a comparative study on the PFSS and GFSS problems. In this study, three metaheuristics were proposed for solving the GFSS and PFSS problems based on two different neighbourhood structures. For the PFSS problem, an insertion neighbourhood structure is used, while for the GFSS problem, a critical-path neighbourhood structure is adopted. In addition, the optimal solutions of thirty 5-job 5-machine GFSS and PFSS instances are exactly obtained by the *integer programming* (IP) models using the AMPL CPLEX software. Among the 30 instances tested, 23 of them yield the same optimal solution for both the PFSS and GFSS problems. For the other seven instances, the optimal makespans for the GFSS problems are better than the optimal makespans for the PFSS problem. From a great deal of computational experiments on the benchmark and randomly-generated instances, it is demonstrated that the optimal solution for the PFSS problem is also optimal for the corresponding GFSS result in most cases. And the makespan of the best PFSS schedule found is usually very close to the optimal makespan of the GFSS schedule.

4.3.2 Job-Shop Scheduling

The job-shop scheduling (JSS) problem differs from the flow-shop scheduling (FSS) problem in one important aspect: the flow of machine routing is not unidirectional.

In the JSS problem, for every job, the operations must be processed in a given order, but this order may differ with various jobs.

The JSS problem with the makespan criterion is regarded as one of the hardest problems in combinatorial optimization. An indication of its difficulty is given by the fact that the famous 10-job 10-machine instance formulated for the first time by Muth & Thompson (1963) was exactly solved by Carlier and Pinson (1989), with a branch and bound algorithm that required about 5 hours of computing time on a PRIME 2655 computer.

Solution Techniques for Job-Shop Scheduling

In addition to the exact methods, many simple heuristics in the early stages were proposed through the use of priority rules. Such approaches use a priority rule to select an operation from a set of candidates to be sequenced next. The candidates may be chosen to create a *nondelay* or *active* schedule in which no machine idle time is allowed if operations are available to be processed. Although priority rule heuristics are undemanding in their computational requirements, the solution quality tends to be erratic.

One effective and non-trivial constructive heuristic method is the *shifting bottleneck procedure* (SBP) algorithm proposed by Adams *et al.* (1988). This algorithm builds up and improves a schedule by constructing iterative solutions of a bottleneck single-machine scheduling problem. The detailed procedure of the SBP algorithm will be described for solving the *parallel-machine job-shop scheduling* (PMJSS) problem in Chapter 6.

Various metaheuristic algorithms for the JSS problem with makespan criterion are reviewed in the following.

Simulated annealing algorithms were first proposed by employing the disjunctive graph model. First, Laarhoven *et al*. (1992) suggested the use of the critical-operation transpose neighbourhood. Yamada *et al*. (1996) used the smaller critical neighbourhood structure. Their algorithm backtracks to the best schedule that is

currently generated whenever certain moves are accepted that do not improve the best solution value, and resets the temperature appropriately. Computational results of the above methods show that the simulated annealing algorithm generates better-quality solutions than those of the shifting bottleneck procedure.

Tabu search (TS) provided an attractive alternative to simulated annealing for the JSS problem. Taillard (1994) adopted the critical path neighbourhood that was used by Laarhoven *et al*. (1992). Special features of Taillard algorithm include quick lower bound estimates instead of exact computation for neighbouring solutions; a dynamic tabu list length that changes randomly after specified numbers of iterations are performed. Barnes and Chambers (1995) used the framework of Taillard's method to design an alternative tabu search algorithm. Dell'Amico and Trubian (1993)'s tabu search used a composite neighbourhood based on the critical path. Nowicki and Smutnicki (1996)'s tabu search also used the block structure on the critical path.

Genetic algorithm (GA) which uses a variety of different representations has been proposed for the JSS problem. Several of these genetic algorithms use a heuristic-based representation: a solution is constructed from its representation by heuristic methods. Storer *et al*. (1992) proposed a data perturbation representation, in which a schedule is constructed from the perturbed data by using SPT as a priority rule to construct a schedule. There are also some genetic algorithms that do not rely on a heuristic-based representation. For example, Nakano and Yamada (1991) used a type of ordered pair representation, and employ a standard crossover operation on the resulting binary strings. Shi (1997) presented a new encoding scheme by applying a matrix to denote the ordered machine numbers. Actually, the main differences of other GA alternatives are the representation methods. The performance of the GA-type algorithms varies case by case. Although comparing favourably with simulated annealing, it cannot compete with the best tabu search algorithm.

Recently, the new types of metaheuristics such as the *ant colony system* (AOC) and *particle swam optimisation* (PSO) algorithms are introduced and become popular in the areas of Operations Research and Combinatorial Optimisation. For example,

Udomsakdigool and Kachitvichyanukul (2008) proposed a multiple AOC algorithm that takes inspiration from the foraging behaviour of a real *ant colony* to solve the JSS problem. In the AOC algorithm, *ants* cooperate to find good solutions by exchanging information among *colonies* which are stored in a master pheromone matrix that serves the role of global memory. The exploration of the search space in each colony is guided by different heuristic information. The proposed algorithm is tested over a set of benchmark problems and the computational results demonstrate that the multiple AOC performs well on the benchmark problems.

Most algorithms for the JSS problem have been tested on the benchmarks generated by Fisher and Thompson (1963), Lawrence (1984), and Taillard (1993). On the evidence of the computational results, the above TS-type algorithms are generally superior to SA-type, GA-type and SBP-type algorithms.

In summary, there emerge some common performance features of metaheuristic algorithms for the JSS problem with makespan criterion. First, it is beneficial to restrict the neighbourhood moves so that only *critical* operations are transposed, swapped, or inserted. Second, to reduce computation time, it is preferable to estimate the makespan rather than perform an exact computation. Third, the possibility of backtracking to a previous solution, rather than always continuing the search from the current solution, allows the search to be redirected to essential parts of the solution space with a higher chance of finding a good quality solution.

4.3.3 Open-Shop Scheduling

The flow-shop and job-shop scheduling problems are widely used in the modeling of industrial production processes. Thus, most of published results in the literature are focused on the FSS and JSS problems.

A rare situation that is related to the OSS problem occurs when a number of processing plants are available. An open shop thus corresponds to a problem in which there are no precedence relationships between operations as each operation can be independently performed by any one plant.

The problem of minimising the makespan in an open shop can be stated as follows. There are a set of n jobs that have to be processed on a set of m machines. Every job consists of m operations, each of which must be processed on a different machine for a given amount of time. At any time, at most one operation can be processed on each machine, and at most one operation of each job can be processed. The operations of each job can be processed in any order, which implies that the machine sequence of every job is immaterial and up to the scheduler to decide. All operation must be processed without interruption. The problem is to find a schedule of the operations that minimise the makespan.

Gonzalez and Sahni (1976) derived a polynomial time algorithm for a two-machine OSS problem and indicated that the three-machine OSS problem is NP-hard in the ordinary sense. There appears no literature work on enumerative algorithms until the middle of the 1990s.

Pinedo (1995) presented a dispatching rule of LPT (*Longest Processing Time*), which solves the two-machine OSS problem in polynomial time. Some OSS cases with special structures are still polynomially solvable. For example, for a three-machine OSS case, Chen and Strusevich (1993) propose a linear time algorithm that transforms a dense schedule into a new schedule such that the obtained makespan is at most 1.5 times worse than the optimal value.

Brucker *et al.* (1997) proposed a branch-and-bound algorithm for the m-machine OSS problem.

There are relatively few heuristic algorithms for the m-machine OSS problem in the literature. Brasel *et al.* (1993) developed one effective constructive insertion algorithm for the m-machine OSS problem. Ramudbin and Marier (1996) extended the *shifting bottleneck procedure* (SBP) algorithm initially proposed by Adams *et al.* (1988) for the JSS problem to solve the OSS problem.

In addition, some metaheuristics have been developed to solve the m-machine OSS problem, including *tabu search* by Alcaide *et al.* (1997) and Liaw (1999b), *hybrid*

genetic algorithm by Fang *et al.* (1994), Prins (2000) and Liaw (2000), and *simulated annealing* by Liaw (1999a). From the computational results, Liaw's algorithms perform extremely well on both benchmarks and randomly generated OSS problems.

4.3.4 Mixed-Shop Scheduling

As described in the previous three sections, there are three main types of multi-stage production environments in practice, namely, *flow shop*, *job shop* and *open shop*. In a flow shop, the operations of each job have to be processed in the unidirectional order. In a job shop, which is a generalization of the flow shop, the machine sequence is given for each job, but different jobs may have different routes. In an open shop, the machine sequence for every job is unrestricted.

In fact, a mixture of the above three "pure" shops is called a *mixed shop*. In a mixed shop, the machine sequences are fixed for some jobs as in a flow shop or in a job shop, while the machine sequences of the other jobs are unrestricted as in an open shop. Problems of this type create a relatively new direction of research because the diversified machine sequences for various jobs often arise in real-world production environments. However, because of strong NP-hardness and complicated characteristics, the mixed-shop scheduling (MSS) problem has just been introduced and has received little attention in the literature.

The term "mixed shop" for this more general type of a multi-stage system was first introduced by Masuda *et al.* (1985), who initiated the theoretical investigation of the MSS problem. In this paper, a mixture of a *flow shop* and an *open shop* was called a mixed shop. They proposed a very complex polynomial-time algorithm to construct a feasible schedule for the two-machine MSS problem with makespan criterion.

Their algorithm was further improved by an algorithm of Strusevich (1991), developed for the two-machine MSS problem in which the mixed shop is a mixture of a *job shop* and an *open shop*.

Shakhlevich *et al.* (1999) proved that the preemptive MSS problem with three machines and three jobs (two jobs have fixed machine sequences and one has an arbitrary machine sequence) is NP-hard. They answered some remaining open questions on computational complexity of the MSS problems. They also presented some polynomial and pseudo-polynomial algorithms for solving some special MSS cases.

Shakhlevich *et al.* (2000) surveyed recent results about the computational complexity of some MSS problems. The main attribution of this paper was devoted to establishing the boundary between polynomially solvable and NP-hard MSS problems.

As the MSS problem consists of more than one type of pure shops, it is usually much more difficult to solve by the exact algorithm than a pure shop. For example, the algorithms of Masuda *et al.* (1985) and Strusevich (1991) for the two-machine MSS problem are much more sophisticated than those for the two-machine FSS problem (Johnson, 1954) and the two-machine OSS problem (Gonzalez and Sahni, 1976). Moreover, to solve the MSS problems with makespan criterion, one needs either to use a mixture of techniques and algorithms developed for solving the "pure" shops, or to develop other algorithms especially designed for this class of problems.

To the best of our knowledge, there are very few articles devoted to the MSS problem. To the best of our knowledge, no published papers on exact algorithms and metaheuristic algorithms for the MSS problem appeared in the literature.

Liu and Ong (2004) initially proposed three metaheuristics (i.e. *Simulated Annealing*, *Threshold Accepting*, and *Tabu Search*) for solving the MSS problem, based on a special neighbourhood structure and an efficient local search procedure. The performance of the proposed methodology was evaluated by means of a set of Lawrence's benchmark data for JSS, and a set of randomly generated data for OSS, and a combination of JSS and OSS test data for MSS. The computational results show that the proposed methodology is very efficient and effective at solving the

MSS problems. In addition, the proposed methodology is very generic because it is also able to solve three "pure" shop scheduling problems with little modification of the test data format. The extensive computational experiments reveal that the MSS problem is relatively easier to solve than the JSS problem due to the fact that the scheduling procedure becomes more flexible by the inclusion of more open-shop-type jobs in the mixed shop. The details of this methodology will be described and illustrated in Section 4.4.

4.3.5 Group-Shop Scheduling

In 1997, the group shop scheduling (GSS) problem was first introduced in the context of a mathematical competition organised by the Technische Universiteit (TU) Eindhoven, Netherlands. The problem was to organise a meeting between parents and teachers, where the length of the conversations is given and where the order of the conversations per parent is almost fixed. The contest was won by Paul Hoogendijk, who found a schedule with a length of 469, which was proved as the best solution by a branch-and-bound search over all potential schedules of length 468. In this search 19509 nodes are evaluated, and the computations take approximately 65 hours on a Pentium II 266 MHz computer. For more information about the features of this problem, please refer to the website: http://www.win.tue.nl/whizzkids/1997.

The GSS problem can be described as follows. Given a set of operations O that is partitioned into two subsets $\Gamma = \{\Gamma_1, \ldots, \Gamma_n\}$ and $\Lambda = \{\Lambda_1, \ldots, \Lambda_m\}$, where n is the number of jobs and m is the number of machines. Γ_i is the set of operations which belong to job i ($i = 1, 2, \ldots, n$). Λ_k is the set of operations that are processed on machine k ($k = 1, 2, \ldots, m$). For each job i, the operations in Γ_i are further partitioned into g groups $G = \{G_1, \ldots, G_g\}$. The operations in the same group are unrestricted while operations in two distinct groups satisfy the precedence relationship between two groups.

If for every $\Gamma_i (i = 1, 2, \ldots, n)$, the number of groups in Γ_i is equal to 1 (i.e. $g = 1$), the GSS problem is equivalent to an *OSS* problem. If for every $\Gamma_i (i = 1, 2, \ldots, n)$, each group in Γ_i consists of only one operation (i.e. $g = m$), this GSS problem turns out to be a *JSS* or *FSS* problem. When $g = 1$ holds for some jobs and $g = m$ holds for other jobs, the GSS problem is treated as a *MSS* problem.

To our knowledge, very few articles in the literature address the GSS problem. However, the GSS problem is the most generalised case that contains the characteristics of the FSS, JSS, OSS and MSS problems. As a result, the methodology proposed for solving the MSS problem can also be applied to the GSS problem with some modification.

For more details, please refer to Liu *et al.* (2005b) who initially proposed a fast tabu search for the GSS problem.

4.3.6 Dynamic Shop Scheduling

The class of shop scheduling problems can be further divided into two categories. One category concerns the *static* shop scheduling problems, in which machine environments and job characteristics are assumed to be known and unchanged. Consequently, it is reasonable to suppose that the five problems discussed above (i.e. FSS, JSS, OSS, MSS or GSS) fall into the first category.

The other category consists of production environments that are *dynamic* in nature. Over the last decade, the crisis in the semiconductor industry (especially in Singapore) indicated that dynamically optimising the operational control of wafer fabrication facilities is vital to success. In this situation, the job characteristics and machine environments may change with time and consequently the planner can not have full control over them. In the literature, very few researchers suggested a comprehensive scheduling model and solution to the dynamically operational control in multi-stage production systems. In addition, most researchers in

scheduling focused on the *static* problems and seldom took into account the *dynamic* disturbance.

To deal with dynamic scheduling, most researchers usually partitioned the scheduling process into two phases. In Phase 1, they consider the optimisation of the makespan under idealised conditions; then in Phase 2, in the case of an accidental disturbance, they simply deal with reactive scheduling based on some dispatching rules. In the following, we review some research that is related to dynamic scheduling.

The first study in this area was initialised by Holloway and Nelson (1974) who implemented a multi-pass procedure in a job shop by generating schedules periodically. They concluded that a periodic policy (scheduling/rescheduling periodically) is effective in dynamic job-shop environments. Muhleman *et al.* (1982) analysed the periodic scheduling policy in a dynamic and stochastic job-shop system. Their experiments suggested that more frequent revision is needed to obtain better scheduling performance. Church and Uzsoy (1992) considered periodic and event-driven rescheduling approaches in a single-machine production system with dynamic job arrivals. Their results indicate that the performance of periodic scheduling deteriorates as the length of the rescheduling period increases and event-driven methods achieve a reasonably good performance. Sabuncuoglu and Karabuk (1997) proposed several reactive scheduling policies to cope with machine breakdowns and processing time variations. Their results indicate that it is not always beneficial to reschedule the operations in response to every unexpected event and the periodic response with an appropriate length can be quite effective in dealing with the interruptions. Subramaniam *et al.* (2000) demonstrated that significant improvements to the performance of dispatching in a dynamic job shop could be achieved through the use of simple machine selection rules. In addition, the reactive scheduling problems have also been studied by implementing knowledge-based, finite-state-machine and other artificial intelligence techniques (Dutta, 1990; Szelke and Kerr, 1994).

Liu *et al.* (2005a) analysed the characteristics of the dynamic mixed-shop scheduling problem when machine break and new job arrivals occur. In a

disjunctive graph model, they innovatively presented a framework to model the ***dynamic mixed-shop scheduling*** problem as a ***static group-shop-type scheduling*** problem. Using the proposed framework, they adapted a fast tabu search metaheuristic once proposed for the five static shop scheduling (i.e. FSS, JSS, OSS, MSS and GSS) problems to a number of *dynamic shop scheduling* cases. The computational results show that under this framework, the proposed methodology can efficiently solve the dynamic shop scheduling problems. In Section 4.4, the description of this methodology will be given in detail.

4.4 Methodology for Multi-Stage Systems

In this section, a state-of-the-art methodology (Liu and Ong 2002; Liu and Ong; 2004; Liu *et al.* 2005a; Liu *et al.* 2005b) that has been successfully applied to the above five types of *static* shop scheduling problems (i.e. FSS, JSS, OSS, MSS and GSS) and *dynamic shop scheduling* (DSS) problems will be presented. In general, there are four main components in this methodology.

The graphical representation for modelling the problems is the first essential part. For the *classical* shop scheduling problems, it is convenient to represent them in terms of the disjunctive graph model.

An efficient algorithm to compute the values of starting times and delivery times is the second important part. In this methodology, a topological-sequence algorithm initially proposed by Liu and Ong (2002) is employed to compute the values of the starting time and delivery time for each node in the disjunctive graph, and then to determine the critical path, the blocks and the groups for constructing the neighbourhood.

The third part of this methodology is to construct the desirable neighbourhood structure according to the given type scheduling problem. Through a great deal of computational experiments, we found that the neighbourhood structure plays a crucial role in the search procedure. If the neighbourhood structure is powerful and

precise enough, it will be fast to find a solution close to the optimum after a finite number of iterations.

The final part is how to design the procedure of metaheuristics, which actually is an art to make a balance between diversification and intensification. As we know, one of the disadvantages for simple local search methods is that they may get trapped easily at a local optimum, which may be far away from the global optimum. From the mid-1980s, some metaheuristics such as *simulated annealing*, *genetic algorithm* (GA), *tabu search* (TS) and *threshold accepting* (TA), have been proposed by many researchers to overcome this disadvantage in many combinatorial optimisation problems.

4.4.1 Disjunctive Graph Models

In this section, a discussion of how to represent and analyse the *flow-shop scheduling* (FSS), *job-shop scheduling* (JSS), *open-shop scheduling* (OSS), *mixed-shop scheduling* (MSS) and *group-shop scheduling* (GSS) problems in terms of disjunctive graph models will be presented. In particular, it will be discovered that the *dynamic mixed-shop scheduling* (DMSS) problem can be converted to be a *static group-shop-type scheduling* problem with the help of a disjunctive graph model.

At first, the *static mixed-shop scheduling* (SMSS) problem is described as follows. Given m machines and n jobs, the n jobs are to be processed on the m machines and each job consists of m operations processed on m machines. In a mixed shop, some jobs (the *flow-shop-type* or *job-shop-type* jobs) have prescribed machine sequences through the m machines, while for other jobs (the *open-shop-type* jobs), their machine sequences through the m machines is unrestricted and forms part of the decision process. The objective function is to minimise the makespan.

For illustration, consider a static JSS instance with 5 machines and four jobs consisting of three *flow-shop-type* jobs and one *job-shop-type* job; and a static MSS instance with 5 machines and four jobs that consist of one *open-shop-type* job, one

job-shop-type job and two *flow-shop-type* jobs. For comparison, two disjunctive graphs are respectively modelled for the JSS and MSS instances, drawn in Figures 4-4 and 4-5.

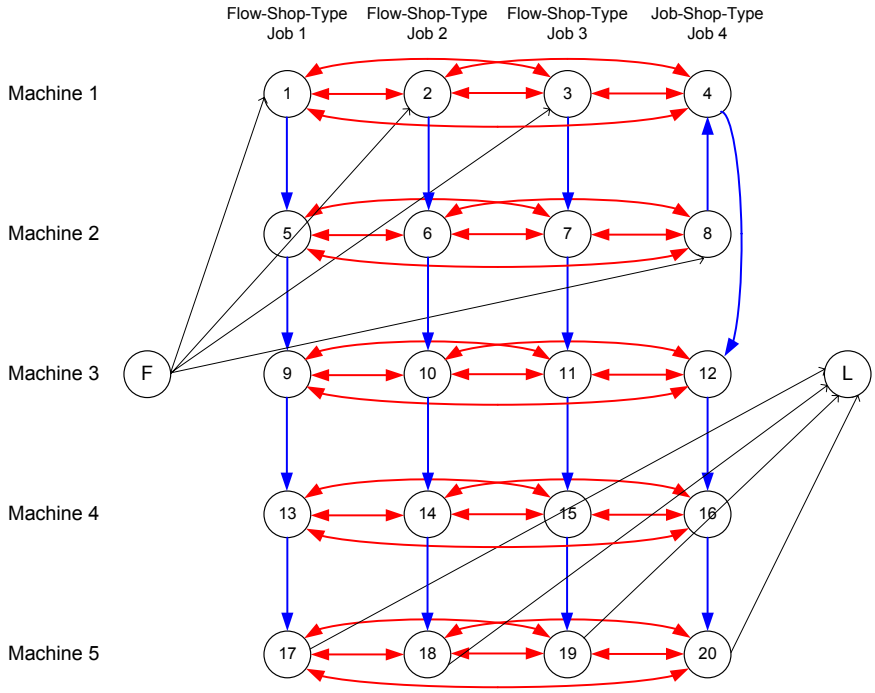

Figure 4-4: Disjunctive graph for a static 5-machine 4-job JSS instance

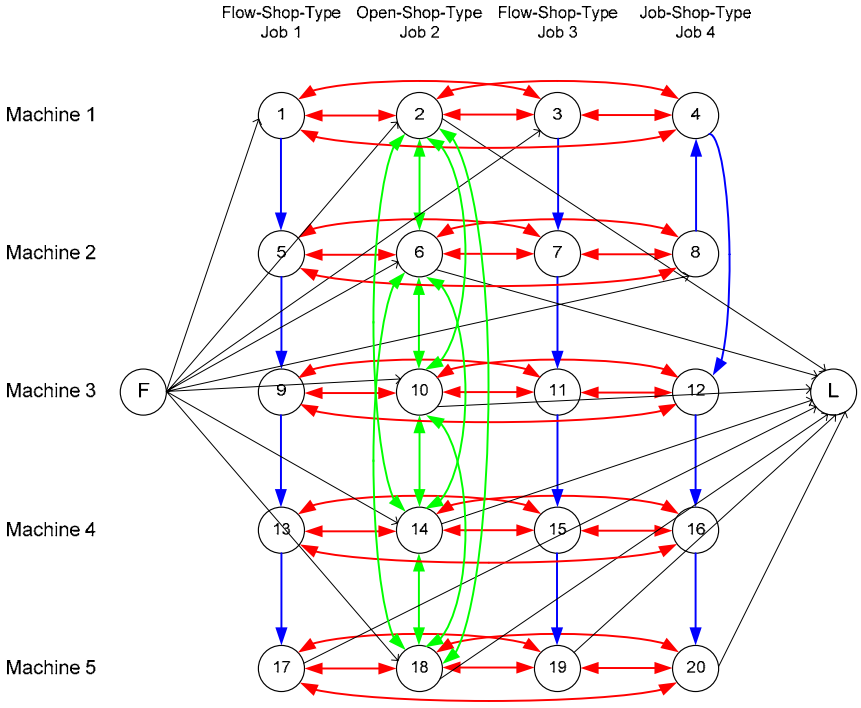

Figure 4-5: Disjunctive graph for a static 5-machine 4-job MSS instance

In a disjunctive graph, the nodes are numbered from 1 to n_{Oper}, where $n_{Oper} = n \times m$ denotes the total number of job operations. Additionally, two dummy nodes are added. Dummy node F connects the operations that are the first ones to be processed. Similarly, dummy node L connects the operations that are the last ones to be processed. Other than these two dummy nodes, every node has at most two immediate predecessors and two immediate successors. An arc (i, j) connects operations i and j, which implies that operation i has to be processed immediately before operation j. Each arc (i, j) has a length equal to p_i, the processing time of operation i.

For each pair of job operations (i, j) that belongs to a *flow-shop-type* (or *job-shop-type*) job, there is only one fixed direction because the operations of every job in a flow shop (or job shop) must be processed in the given order. These arcs are called ***conjunctive arcs***. Conversely, for each pair of operations (i, j) processed on the same machine or belonging to an *open-shop-type* job, there are two direction choices, i.e. (i, j) and (j, i). These arcs with two opposite direction choices are called ***disjunctive arcs***.

In addition, for each node i which is either the operation of an *open-shop-type* job or the first operation of a *flow-shop-type* (or *job-shop-type*) job, a virtual conjunctive arc (F, i) with the length of zero is added to the graph. Similarly, for each node i, which is the operation of an *open-shop-type* job or the last operation of a *flow-shop-type* (or *job-shop-type*) job, a virtual conjunctive arc (i, L) with the length of p_i is drawn. Therefore, a feasible schedule can be obtained by choosing the direction choice among all disjunctive arc pairs, in such that the schedule is *acyclic*.

If the redundant disjunctive arcs in Figure 4-5 are removed, a feasible MSS schedule can be obtained and drawn in Figure 4-6.

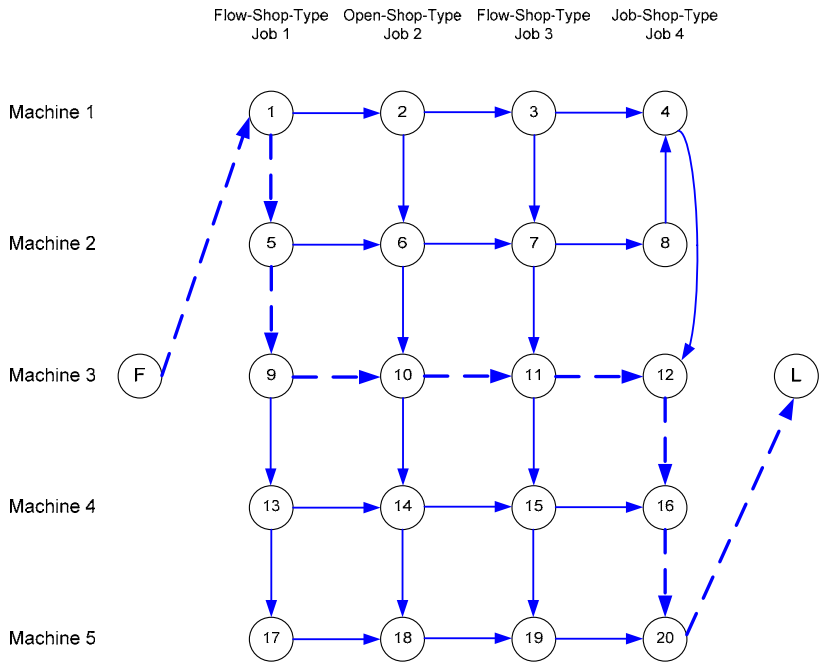

Figure 4-6: Disjunctive graph for a feasible schedule of the SMSS instance, in which a critical path is highlighted by thicker dot line

In this case, the complete set of directed arcs is presented as follows:

- Job 1: 1→5, 5→9, 9→13, 13→17;
- Job 2: 2→6, 6→10, 10→14, 14→18;
- Job 3: 3→7, 7→11, 11→15, 15→19;
- Job 4: 8→4, 4→12, 12→16, 16→20;
- Machine 1: 1→2, 2→3, 3→4;
- Machine 2: 5→6, 6→7, 7→8;
- Machine 3: 9→10, 10→11, 11→12;
- Machine 4: 13→14, 14→15, 15→16;
- Machine 5: 17→18, 18→19,19→20;
- Dummy First Node: F→1;
- Dummy Last Node: 20→L;

Table 4-2: The processing times of operations in a MSS instance

	Job 1	**Job 2**	**Job 3**	**Job 4**
Machine 1	1	1	1	1
Machine 2	2	2	2	2
Machine 3	3	3	3	3
Machine 4	1	1	1	1
Machine 5	2	2	2	2

To give a clearer picture of the schedule, a Gantt chart as shown in Figure 4-7 can be used to display the start and end times of each operation, if the processing times of operations for this MSS instance are given in Table 4-2.

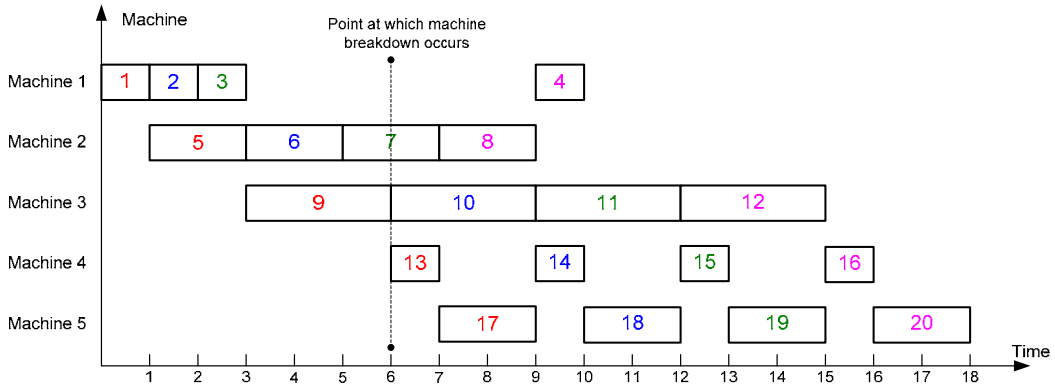

Figure 4-7: The Gantt chart for a feasible MSS schedule

Based on the above definition of the *static* MSS (SMSS) problem, the *dynamic* MSS (DMSS) problem can be stated as follows. In a mixed-shop production environment, how should the jobs be scheduled and how should they be reactively scheduled when dynamic events occur, so that the makespan is dynamically minimised? As reported in the literature (Ramasesh 1990), there are a number of factors that characterise a dynamic scheduling problem. In this study, only two typical dynamic factors are considered, that is, *machine breakdown* and *new job arrivals*. Using this example, now we can discuss how to analyse and reactively model the DMSS problem by the disjunctive graph.

Machine Breakdown

In the event of machine breakdown, the scheduling process has to be dynamically partitioned into two phases: *static scheduling phase* and *reactive scheduling phase*. In Phase 1, the operations are processed according to the optimal or best schedule obtained under static conditions; then, in Phase 2, the reactive scheduling is dealt with in the event of machine breakdown.

In this case, there are three main parameters that can influence the reactive scheduling phase. The first parameter is the time point t at which the machine

breakdown occurs. The second parameter is the machine k that has broken down. The third parameter is L, the length of the breakdown or the time taken to repair the machine. In addition, two scenarios should be considered: semifinished job operations to be resumed or the entire job to be taken out from the schedule. For the first case, the unfinished operation usually has priority to be processed first when the machine has been repaired, considering the set-up time or other realistic factors. For the second case, the affected job should be taken out either to be discarded or processed offline.

For illustration, we assume that the feasible schedule shown in Figure 4.4 is the best schedule obtained under static conditions; Machine 2 is broken down at time point $t = 6$, and will be repaired in three time units. In other words, the three main parameters are defined as: $t = 6$, $k = 2$, $L = 3$.

If the semifinished job operations are to be resumed after the repair of the machine, then the reactive scheduling phase could be remodelled by the disjunctive graph shown in Figure 4-8.

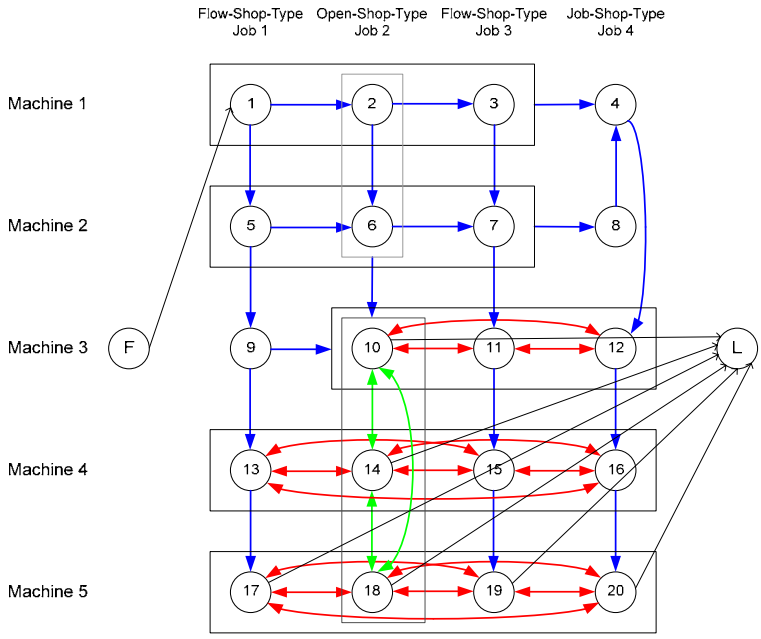

Figure 4-8: Disjunctive graph for reactive scheduling phase when machine breakdown occurs and the affected jobs are to be resumed after the repair of the machine

As shown in the Gantt chart of Figure 4-7, at time $t = 6$, Operations 1, 2, 3, 5, 6, and 9 have been processed; Operation 7 is in the process and has not been

completed yet. After the repair of Machine 2, Operation 7 will continue to be processed. In this case, the processed and unprocessed 20 operations can be divided into various groups as illustrated in Figure 4-8.

Similarly, if the affected jobs are to be discarded, in this case the unfinished job, Job 3 to which Operation 7 belongs, should be taken out from the schedule. The 15 remaining operations can be divided into several groups and the reactive scheduling phase can then be carried out from the new disjunctive graph, as illustrated in Figure 4-9.

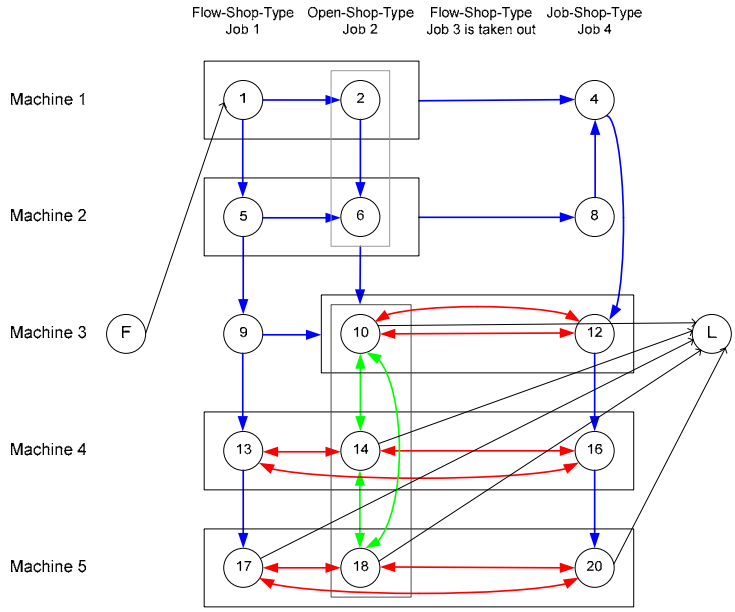

Figure 4-9: The disjunctive graph for reactive scheduling phase when machine breakdown occurs and the affected jobs are taken out of the schedule

From the analysis of Figures 4-8 and 4-9, it is discovered that the *dynamic* shop scheduling problem can be converted to be a special case of the *static* group-shop-type scheduling problem. For those operations that have been processed before the time point of machine breakdown, their arc directions are fixed, while for those that remain to be processed, their arc directions may be unrestricted. It is noted that not only two distinct groups of the same open-shop-type job but also two distinct groups on the same machine satisfy the precedence relationship.

New job arrivals

In the event of new job arrivals, the scheduling process is also partitioned into some phases which depend on how frequently the new jobs will arrive. The schedule is

optimised under static conditions in the first phase. In this scenario, there are two main parameters that influence the second reactive scheduling phase. The first parameter is the time point t at which the new jobs arrive. The second parameter is the number of jobs to arrive.

For illustration, using this same SMSS data given in Table 4-2, we assume that in the first phase, there are only two jobs, Jobs 1 and 2, to be processed. The disjunctive graph for a feasible two-job SMSS schedule constructed in Phase 1 is given in Figure 4-10.

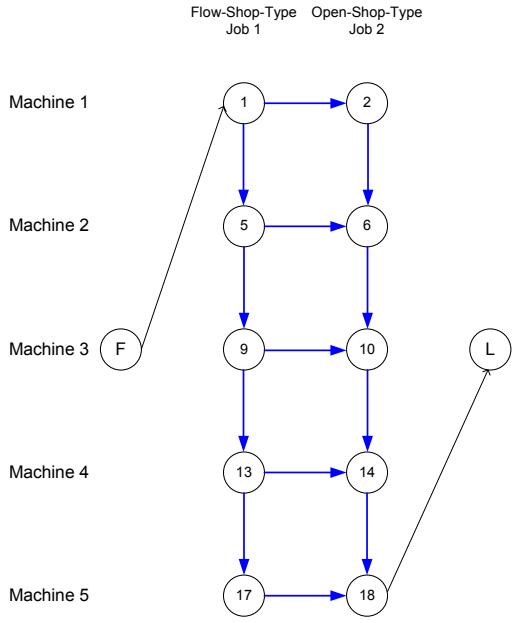

Figure 4-10: The disjunctive graph for the schedule constructed in Phase 1 with two jobs processed

At $t = 6$, another two jobs, Jobs 3 and 4, have arrived in the production environment. From the Gantt chart shown in Figure 4-7, at $t = 6$, five operations (i.e. Operations 1, 2, 5, 6 and 9) have been processed and the other five operations (i.e. Operations 10, 13, 14, 17 and 18) have not been done yet. Accordingly, the disjunctive graph for these four jobs in the reactive scheduling phase can be reconstructed and drawn in Figure 4-11.

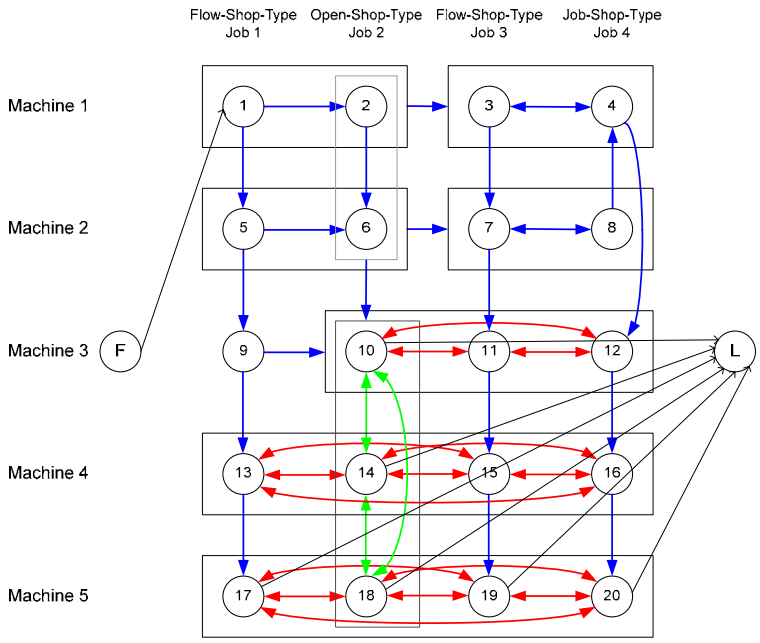

Figure 4-11: The disjunctive graph for reactive scheduling with additional two jobs arrival in Phase 2

In this scenario, it is observed that this DMSS case with new job arrivals can also be modelled as a static group-shop-type scheduling problem.

From the above discussions, it is validated that the disjunctive graph is a very useful tool to model and analyse the classical shop scheduling problems.

4.4.2 Topological-Sequence Algorithm

For any feasible schedule, every node (except the two dummy nodes F and L) of the corresponding graph has at most two immediate predecessors and two immediate successors. A critical path contains an ordered sequence of operations from node F to node L. Operation i is *critical* if and only if

$$e_i + p_i + l_i = C_{\max} \tag{4.3}$$

where e_i, p_i, and l_i are respectively the stating times, processing times, and delivery times of Operation i.

Furthermore, the critical path can be decomposed into some subsets of operations, called *blocks*. A block is a subset of consecutive operations that contains operations

belonging to the same job or operations processed on the same machine. Note that a block must have at least two operations. For any two adjacent blocks, one block consists of operations of the same job and the other block consists of operations processed on the same machine. For example, the critical path *F*-1-5-9-10-11-12-16-20-*L* shown in Figure 4-6 can be decomposed into three blocks (parts): 1-5-9, 9-10-11-12 and 12-16-20.

The approach to calculate the starting times and delivery times for each operation and then to determine the critical path and blocks is the key step to solve the classical multi-stage scheduling problems. In the literature, the algorithm of Taillard (1994) or the algorithm of Bellman (1958) can be applied to calculate these values. In this methodology, a topological-sequence algorithm is proposed by Liu and Ong (2002) to calculate these values.

Topological-Sequence Algorithm

Step 1: Compute the in-count value (the number of predecessors) of each node.

Step 2: Find a topological sequence as follows:

2.1 Select node *F* as the first node on the topological order list.

2.2 Decrement the in-count value for each of the immediate successor nodes of the selected node by 1.

2.3 Select any of the unselected nodes having a zero in-count value and put this node as the next node on the topological order list.

2.4 Repeat Steps 2.2 and 2.3 until all nodes are selected.

Step 3: Starting from setting $e_0 = 0$, calculate the e_i values of all nodes in the topological sequence according to

$$e_i = \max\{e_{PM[i]} + p_{PM[i]}, e_{PJ[i]} + p_{PJ[i]}\} \tag{4.4}$$

where $PM[i]$ is the same-machine operation processed just before operation i, if it exists; $PJ[i]$ is the same-job operation that just precedes operation i, if it exists;

Step 4: Starting from setting $l_{n_{Oper}+1} = 0$, calculate the l_i of all the nodes in the reverse order of the topological sequence according to

$$l_i = \max\{l_{SM[i]} + p_{SM[i]}, l_{SJ[i]} + p_{SJ[i]}\} \tag{4.5}$$

where $SM[i]$ is the same-machine operation processed just after operation i, if it exists; $SJ[i]$ is the same-job operation that immediately follows operation i, if it exists;

4.4.3 Neighbourhood Structure

In a local search procedure, the definition of neighbourhood structure has a great effect on the efficiency and effectiveness of the algorithm. Different neighbourhood structures have different properties, which may make them more or less suitable depending on the particular type of problem to be solved.

Definition of Candidate Arcs

Given a feasible schedule, in order to improve its solution, we have to change the sequence of operations on the critical path. Actually, we only need to focus on considering the arc (i, j) of the blocks, where at least one of arcs $(PM[i], i)$ and $(j, SM[j])$ is not on the critical path. In other words, either operation i is the first or j is the last one in a block. These arcs are called ***candidate arcs***. This restriction is motivated by the fact that the swap of a non-candidate arc (i, j) cannot improve the makespan. The proof of this property is given as follows (Liu and Ong 2002).

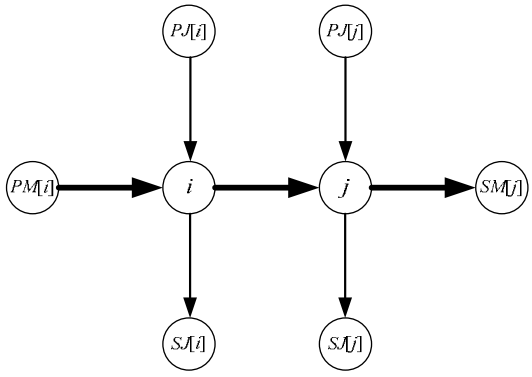

Figure 4-12: The property analysis of a candidate arc in the disjunctive graph

As shown in Figure 4-12, if the arcs of (i, j), $(PM[i], i)$ and $(j, SM[j])$ are on a critical path, thus we have such relationships:

$$e_i = e_{PM[i]} + p_{PM[i]}$$

$$l_j = l_{SM[j]} + p_{SM[j]}$$

$$e_i + p_i + p_j + l_j = C_{\max}.$$

After swapping (i, j) to (j, i), the new stating times of operations i and j are:

$$e_j^{/} = \max\{e_{PJ[j]} + p_{PJ[j]}, e_{PM[i]} + p_{PM[i]}\} = \max\{e_{PJ[j]} + p_{PJ[j]}, e_i\},$$

$$e_i^{/} = \max\{e_{PJ[i]} + p_{PJ[i]}, e_j^{/} + p_j\}$$

which implies that:

$$\because e_j^{/} \geq e_i \text{ and } e_i^{/} \geq e_j^{/} + p_j$$

$$\therefore e_i^{/} \geq e_i + p_j$$

In the same fashion, the new delivery times of operations i and j are:

$$l_i^{/} = \max\{l_{SJ[i]} + p_{SJ[i]}, l_{SM[j]} + p_{SM[j]}\} = \max\{l_{SJ[i]} + p_{SJ[i]}, l_j\}$$

$$l_j^{/} = \max\{l_{SJ[j]} + p_{SJ[j]}, l_i^{/} + p_i\},$$

which also implies that:

$$\because l_i^{/} \geq l_j \text{ and } l_j^{/} \geq l_i^{/} + p_i$$

$$\therefore l_j^{/} \geq l_j + p_j$$

Hence, according to the above relationships, it is proved that the new solution is worse than the current one:

$$\begin{aligned} Z &= \max(e_i^{/} + p_i + l_i^{/}, e_j^{/} + p_j + l_j^{/}\} \\ &\geq \max\{e_i + p_j + p_i + l_j, e_i + p_j + l_j + p_i\} \\ &\geq C_{\max} \end{aligned}$$

where Z denotes the length of the new longest path which passes through one of the nodes i or j, or both of them, after swapping (i, j) to (j, i).

Therefore, the swap of a non-candidate arc (i, j) when both $(PM[i], i)$ and $(j, SM[j])$ are on the critical path, cannot improve the makespan of the current schedule directly. The neighbourhood structure plays a very important role in solving the problem fast and optimally. If the neighbourhood structure is powerful enough, the algorithms can construct a finite sequence of moves, which will lead from an initial solution to a globally minimal solution. Note that this property does not hold for a limited or inefficient neighbourhood structure.

As the mixed-shop scheduling (MSS) problem is a generalisation of the flow-shop scheduling (FSS), job-shop scheduling (JSS) and open-shop scheduling (OSS) problems, we present the entire neighbourhood structure suitable for the MSS problems. The entire neighbourhood structure is divided into two main types called ***Neighbourhood* 1** and ***Neighbourhood* 2** in terms of the characteristics of a candidate arc (i.e., whether the two operations of a candidate arc are processed on the same machine or belong to the same job).

Neighbourhood 1

Let $b = (b', x, b'')$ denote a block, where b' and b'' are two subsets of b. When the operations of a candidate arc (i, j) and the operations in block b are all processed on the same machine, the following nine possible neighbourhood moves are considered:

(1) Change b to (x, b', b'').

(2) Change b to (b', b'', x).

(3) Change the set of arcs ($PM[i]$, i, j) to (j, $PM[i]$, i) if $PM[i]$ exists

(4) Change the set of arcs ($PM[i]$, i, j) to (j, i, $PM[i]$) if $PM[i]$ exists

(5) Change the set of arcs (i, j, $SM[j]$) to (j, $SM[j]$, i) if $SM[j]$ exists

(6) Change the set of arcs (i, j, $SM[j]$) to ($SM[j]$, j, i) if $SM[j]$ exists

(7) Reverse two arcs (i, j) and $(PJ[j], j)$ at the same time if $PJ[j]$ and j are in the same block (group).

(8) Reverse two arcs (i, j) and $(i, SJ[i])$ at the same time if i and $SJ[i]$ are in the same block (group).

(9) Reverse three arcs (i, j), $(PJ[j], j)$ and $(i, SJ[i])$ at the same time if not only $PJ[j]$ and j but also i and $SJ[i]$ are in the same block (group).

For example, a possible move for Case 2 in Neighbourhood 1 is depicted as

$$PM[PM[i]] \rightarrow PM[i] \rightarrow i \rightarrow j \rightarrow SM[j] \;\Rightarrow\; PM[PM[i]] \rightarrow j \rightarrow PM[\mathrm{i}] \rightarrow i \rightarrow SM[j]$$

and the new values of the following parameters can be sequentially calculated as follows:

$$e'_j = \max\{e_{PM[PM[i]]} + p_{PM[PM[i]]}, e_{PJ[j]} + p_{PJ[j]}\}$$

$$e'_{PM[j]} = \max\{e_{PJ[PM[i]]} + p_{PJ[PM[i]]}, e'_j + d_j\}$$

$$e'_i = \max\{e_{PJ[i]} + p_{PJ[i]}, e'_{PM[i]} + p_{PM[i]}\}$$

$$l'_i = \max\{l_{SJ[i]} + p_{SJ[i]}, l_{SM[j]} + p_{SM[j]}\}$$

$$l'_{PM[i]} = \max\{l_{SJ[PM[i]]} + p_{SJ[PM[i]]}, l'_i + p_i\}$$

$$l'_j = \max\{l_{SJ[j]} + p_{SJ[j]}, l'_{PM[i]} + p_{PM[i]}\}$$

$$Z_2 = \max\{e'_j + d_j + l'_j, e'_{PM[j]} + p_{PM[i]} + l'_{PM[j]}, e'_i + p_i + l'_i\}$$

The value Z_2 gives an estimate for the new makespan, which is the length of the new longest path that passes through at least one of the nodes in the set $\{PM[i], i, j\}$. In this case, a directed cycle can occur if and only if there is a directed path from $SJ[PM[i]]$ to $PJ[j]$, a directed path from $SJ[i]$ to $PJ[j]$, or a directed path from $SJ[i]$ to $PJ[PM[i]$ in the new schedule, illustrated in Figure 4-13.

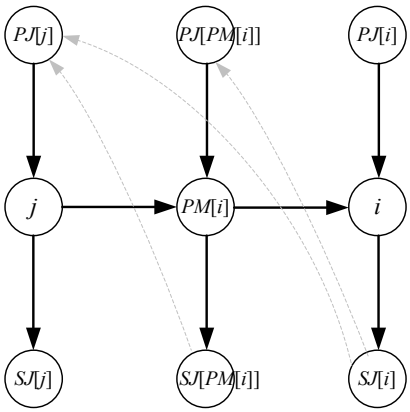

Figure 4-13: The analysis of possible cycles after neighbourhood move of Case 2

Therefore, this move is allowable only if the following inequalities are satisfied:

$e_{PJ[j]} + p_{PJ[j]} \le e_{SJ[PM[i]]}$ if $PJ[j]$ and $SJ[PM[i]]$ exist;

$e_{PJ[j]} + p_{PJ[j]} \le e_{SJ[i]}$ if $PJ[j]$ and $SJ[i]$ exist;

$e_{PJ[PM[i]]} + p_{PJ[PM[i]]} \le e_{SJ[i]}$ if $PJ[PM[i]]$ and $SJ[i]$ exist;

Neighbourhood 2

When the operations of a candidate arc (i, j) and the operations in block b belong to the same job, the following nine possible neighbourhood moves are considered:

(10) Change b to (x, b', b'').

(11) Change b to (b', b'', x).

(12) Change the set of arcs ($PJ[i]$, i, j) to (j, $PJ[i]$, i) if $PJ[i]$ exists

(13) Change the set of arcs ($PJ[i]$, i, j) to (j, i, $PJ[i]$) if $PJ[i]$ exists

(14) Change the set of arcs (i, j, $SJ[j]$) to (j, $SJ[j]$, i) if $SJ[j]$ exists

(15) Change the set of arcs (i, j, $SJ[j]$) to ($SJ[j]$, j, i) if $SJ[j]$ exists

(16) Reverse two arcs (i, j) and ($PM[j]$, j) at the same time if $PM[j]$ exists.

(17) Reverse two arcs (i, j) and (i, $SM[i]$) at the same time if $SM[i]$ exists.

(18) Reverse three arcs (i, j), ($PM[j]$, j) and (i, $SM[i]$) at the same time if both $PM[j]$ and $SM[i]$ exist.

Extensive computational experiments have shown that the proposed neighbourhood structure is very efficient and effective at finding a very good solution in a finite number of iterations. For example, in the Whizzkid97 (20-job 15-machine GSS) instance, the makespan value descends to 496 from the initial makespan value of 3684 in about 5000 iterations (or about 40 seconds). How to apply the above neighborhood structure depends on the characteristics of the given problem type. For example, if the given problem is the OSS or MSS problem, all of the 18 neighbourhood cases are suitable for implementation. For the JSS or FSS problem, only six cases (from Case 1 to Case 6) are appropriate for adoption.

For more details about this neighborhood structure, please refer to the papers (Liu and Ong 2002; Liu and Ong; 2004; Liu *et al.* 2005a; Liu *et al.* 2005b).

4.4.4 Metaheuristic Algorithms

The development of solution approaches for the classical scheduling problems has been the subject of numerous research efforts since the early 1950s. Both optimisation (exact) and approximation (heuristic) algorithms were proposed in the literature. The exact algorithms are mainly based on *Integer Programming* formulations and *Branch and Bound* schemes. However, they are time-consuming, overflow memory, and only solve moderate sized problems optimally within a reasonable time limit. Although considerable progress has been made for exact algorithms, practitioners still find such algorithms unattractive. Therefore, considerable research has been devoted to approximate algorithms, some of which

are not trivial because of a good balance between computational complexity and solution quality.

For a problem in which the objective function is to be minimised, the simplest form of local search is a *descent method* that starts with an initial solution. The descent method should at least have a mechanism for generating a neighbor of the current solution. If the generated neighbor has a smaller objective value, it becomes the new current solution; otherwise the current solution is retained. The process is repeated until a solution is reached with no possibility of improvement in its neighborhood. Such a point is a local minimum and the descent method terminates. By only requiring that the iterative steps move downhill on the objective function surface, they may get stuck at a local minimum which may be far away from the global minimum. This is one of the disadvantages of simple local search methods. Therefore, some metaheuristics such as *simulated annealing* (SA), *tabu search* (TS) and *threshold accepting* (TA) were implemented by researchers in the literature to overcome this disadvantage since the middle of the 1980's.

The following notations will be used in the following subsections to describe these three metaheuristics: SA, TA, and TS:

T_0	an initial temperature in SA or an initial threshold in TA
S	a feasible schedule
$C_{max}(S)$	the makespan value of a schedule
BestVal	the best makespan value found by the algorithm
BetterVal	the better solution found in L iterations at one temperature in SA
max_count	the predetermined maximum *count* number in SA
moves_T	the maximum numbers of neighborhood moves at each temperature or threshold in the SA or TA algorithm
α	the cooling factor for temperature in SA.
$Z(S)$	an estimate for the makespan value of a neighbouring schedule
LB	a lower bound of the makespan

Simulated Annealing

Ever since its introduction by Kirkpatrick *et al.* (1983), the *simulated annealing* (SA) algorithm has been successfully applied to a variety of combinatorial

optimization problems. SA is a type of metaheuristic algorithm to avoid getting trapped at a local minimum by sometimes accepting a neighbourhood move that increases the objective value, using a probabilistic acceptance criterion. These uphill moves make it possible to move away from local minima and explore the feasible region in entirety. The acceptance or rejection of an uphill move is determined by a random acceptance function that may be equal to $\exp(-\delta / T)$, where T is the control parameter called ***temperature*** in the analogy with the physical annealing process, and δ is the difference of objective function values between two successive moves. This acceptance function implies that the moves of small increases in the objective function are more likely to be accepted than large increases.

In addition, the moves are more easily to be accepted when temperature is high. SA should start with a relatively high value of T so that the search is not being prematurely trapped in a local optimum. The algorithm proceeds by conducting a certain number of iterations at each temperature T, while the temperature is gradually dropped. From the computational experiment, the selection of an appropriate high starting temperature T_0 is very important. An annealing schedule can be defined to be a sequence of temperatures such that $T_0 > \alpha T_0 > \alpha^2 T_0 > \ldots > 0$, where α is the reduction factor for temperature T. When T drops to zero, the SA algorithm becomes a simple descent method and will not accept any uphill moves. A full description of the implemented SA approach is presented as follows:

Step 1: Generate an initial schedule *S*.
Apply the topological-sequence algorithm, and then determine the critical path.
Set $BestVal = C_{max}(S)$ and $BestVal = BestVal$.

Step 2: Select an appropriate initial temperature $T = T_0$.

Step 3: Set $count = 0$ and the number of moves to be performed at each temperature T, $moves\_T$.

Step 4: while (($count < max\_count$) and ($BetterVal > LB$))

4.1 Perform the following loop $moves\_T$ times:

4.1.1 Randomly generate a neighbour S^* from S.

4.1.2 Compute $\Delta\delta = Z(S^*) - M(S)$.

If $\Delta\delta < 0$, set $S = S^*$.

Otherwise, generate a random number $R \in (0, 1)$.

Set $S = S^*$ if $R < \exp(-\Delta\delta/T)$.

4.1.3 Generate a new schedule by reversing the selected arcs and apply the topological-sequence algorithm.

4.1.4 If $C_{max}(S^*) < BetterVal$, $BetterVal = C_{max}(S^*)$.

4.2 Set $T = \alpha * T$.

4.3 Calculate *pct*.

If $BetterVal < BestVal$, set $count = 0$ and $BestVal = BetterVal$.

Otherwise, set $count = count + 1$ if $pct < min\_pct$.

Step 5: Return *BestVal* as the best solution.

where *pct* is the percentage of accepted moves in a loop at one temperature, and *min_pct* is the predetermined minimum value which is set as 0.5% in this study.

The initial temperature T_0 must be high enough, allowing many inferior moves to be accepted to explore the broader feasible region. The SA algorithm terminates if an optimal solution has been found (e.g. the solution found equals to the lower bound) or the parameter *count* reaches a predetermined limit, *max-count*. The value of *count* is initialised to zero and incremented by one each time when *pct* is less than a given parameter, *min_pct*; but it can be reset to zero each time if a new best solution in the *moves_T* iterations at a certain temperature is found.

Threshold Accepting

Threshold accepting (TA) was introduced by Dueck and Scheurer (1990) as a deterministic version of simulated annealing. Threshold accepting uses a nonincreasing sequence of deterministic thresholds. Due to the use of positive thresholds, the uphill moves that are accepted in a limited way depend on the threshold values. In the procedure of the TA's execution, the threshold values are gradually lowered, eventually to zero, in which case only improvements are accepted. The essential difference between the SA and TA is the different acceptance rules. An apparent advantage of TA is its simplicity. It is not necessary to compute probabilities or make random decisions. However, one of the major unresolved problems is the determination of appropriate values for the thresholds.

Similar to SA, the initial threshold T_0 must be high enough, allowing many nonimproving moves to be accepted to explore the feasible region in its entirety. The TA algorithm terminates if an optimal solution has been found (the solution equals to *LB*), the threshold value reduces to zero, or the parameter *count* reaches a predetermined limit, *max-count*.

Tabu Search

Tabu search (TS) was initially proposed by Glover (1986). The basic ideas were also sketched by Hansen (1986). Additional efforts of formalisation for TS were reported by Glover (1989, 1990). Many computational experiments have shown that TS has now become an established approximation technique, which can compete with almost all known metaheuristic techniques, and can beat many classical procedures by its flexibility.

TS algorithm is an iterative improvement approach designed to escape from terminating at a local optimum prematurely for combinatorial optimisation problems. Like SA and TA, TS also uses the same neighbourhood structure to move from one region of the search space to another in order to look for a better solution. When a solution is stuck at a local optimum, SA or TA attempts to escape from it by accepting an inferior solution, which may lead to better solutions later. In contrast, TS allows the search to move to the best solution among the set of candidate moves as defined by the neighbourhood structure. However, subsequent iterations may cause the search to move repeatedly back to the same local optimum. In order to avoid cycling to some extent, moves that would return to a recently visited solution should be forbidden for a certain number of iterations. This is accomplished by keeping the attributes of the forbidden moves in a list, called a *tabu list*. The size of the tabu list must be large enough to prevent cycling, but small enough not to forbid too many moves. This systematic use of memory is an essential feature of tabu search.

Additionally, an aspiration criterion is defined to deal with the case in which a move that leads to a new best solution is tabu. If a current tabu satisfies the

aspiration criterion, its tabu status is cancelled and it becomes an allowable move. The feature of tabu lists enables TS to diversify its search to other unexplored regions of the solution space. The use of an aspiration criterion allows TS to lift the tabu restrictions and intensify the search into a particular region of solution space. The stopping rule that ends the search traditionally consists of setting a limit on the execution time, the maximum number of iterations, and the number of consecutive iterations without improving the makespan.

The contrasts between SA (or TA) and TS are fairly obvious. Undoubtedly, the most prominent is the focus on exploiting memory like the tabu list that is absent in SA. In addition, TS emphasizes exploring successive neighborhoods to identify moves of high quality. This contrasts with SA of randomly sampling among the moves by applying an acceptance criterion that disregards the quality of other moves available. In a sense, TS owes its efficiency to a rather fine tuning of an apparently large collection of elements and parameters.

Therefore, the application of TS needs detailed definitions of basic elements including *initial solution*, *move*, *neighbourhood*, *searching strategy*, *memory*, *aspiration function*, *perturbation*, *stopping rules* and values of several tuning parameters such as *the length of a tabu list*, *limit of iterations* and *level of aspiration*. Note that, for the same problem, one can devise various TS-type algorithms which are essentially different with respect to these basic elements.

The detailed steps of TS will be described for solving the train scheduling problems in Chapter 6. Please refer to the papers (Liu and Ong 2002; Liu and Ong; 2004; Liu *et al.* 2005a; Liu *et al.* 2005b) for more details of the implementation and computational results of these three metaheuristics.

4.5 Summary

In conclusion, the graph model for the scheduling problems, the efficient approach to compute the objective function value, the powerful neighborhood structure, and

the state-of-the-art implementation of metaheuristics are four significant components in solving the classical multi-stage scheduling problems.

The different representations of schedule provide many possible avenues to define the neighborhood structure. The approach to compute the makespan and determine the critical path has a great effect on the computation time of the entire algorithm. The powerful neighborhood structure can give a correct search direction to find the better solutions fast. The application of metaheuristics seems like a good guide to direct the local search to move away from local minima and explore the broader feasible region in its entirety.

The MSS and GSS problems receive little attention in the literature. However, they are much more generalised and complicated than three well-known classical FSS, JSS and OSS problems. By appropriate implementation of the above proposed neighbourhood structure, this generic methodology can be applied to efficiently solve these five *static shop scheduling* problems (FSS, JSS, OSS, MSS and GSS) and the *dynamic shop scheduling* problems like DMSS when machine breakdown and new job arrivals occur.

Chapter 5 Non-Classical Scheduling

PUBLICATIONS ARISING FROM CHAPTER 5:

Liu, S. Q., & Kozan, E. (2007b). Scheduling a flow shop with combined buffer conditions. *International Journal of Production Economics* (In Press).

Liu, S. Q., & Kozan, E. (2007a). A topological-sequence algorithm based on alternative graph for the shop scheduling problems with blocking. *Journal of Intelligent Manufacturing* (Second revision submitted).

5.1 Introduction

As defined in Chapter 4, in terms of the capacity of inter-machine buffer storage, the multi-stage scheduling problems are distinguished as ***classical*** or ***non-classical***. If the capacity of buffer storage between any two successive machines is *infinite*, this problem is regarded as a *classical* type problem. Otherwise, it is referred to as a *non-classical* type problem such as *blocking* or *no-wait*.

In recent years, a considerable amount of interest has arisen in the *blocking* and *no-wait* scheduling problems. This interest appears to be motivated both by applications and by research issues. Scheduling problems with *blocking* constraints arise in serial manufacturing processes where no intermediate buffer storage is available. In such situations, a job which has completed processing on a machine may remain there until a downstream machine becomes available, but this prevents another job from being processed there. Scheduling problems with *no-wait* constraints occur in a production environment in which a job (or product) must be processed from start to finish, without any interruption either on or between machines.

One main reason for the occurrence of a *no-wait* or *blocking* production environment is a lack of buffer storage between intermediate machines. In this case, two different environments arise. In *no-wait* scheduling, a job must leave a machine immediately after processing is completed. In comparison, *blocking* scheduling permits a job either to leave immediately after processing if possible or to remain there until the downstream machine on the routing is available. In a sense, a scheduling problem with *blocking* may be thought of as a relaxation of the problem with *no-wait* that is more restrictive.

The other important reason for the occurrence of a *no-wait* or *blocking* production environment lies in the production technology itself or service requirements. In some processes, the temperature or other characteristics of materials require that each operation follows the predecessor immediately. For example, scheduling problems with no-wait often arise in petrochemical production environments and

hot metal rolling industries, where chemicals or metals have to be continuously processed at high temperatures. Another example of a no-wait situation arises in passenger train scheduling because passenger trains should traverse continuously from departure to terminal without any interruption. Scheduling problems with blocking can also be found in the chemical industry, where partially-processed chemical products sometimes must be kept in the machines because of lack of intermediary storage. In addition, blocking situations typically arise in train scheduling problems due to the lack of buffer space. As a consequence, the real-world train scheduling problem should seriously consider blocking or hold-while-wait constraints, which means that the track section cannot release and must hold the train until the next section on the routing becomes available.

In this research, with exploring the properties of blocking conditions based on an *alternative graph* that is an improvement of classical disjunctive graph, a new approach called a *topological-sequence algorithm* is proposed to obtain the feasible solution of the *flow-shop* and *job-shop scheduling* problem with blocking (Liu and Kozan 2007a). The proposed algorithm is generic and adaptive because without any modification it can be used to solve the blocking permutation or general (non-permutation) flow-shop scheduling (BPFSS or BGFSS) problems, and the blocking job-shop scheduling (BJSS) problem. To indicate the preeminence of the proposed algorithm, the results of this study are compared with two known algorithms for BPFSS in the literature.

On the boundary of the no-wait/blocking scheduling environment are a number of interesting issues. In real life, the flow-shop with combined buffer conditions is a more common scheduling environment. Thus, a new non-classic type problem called a combined-buffer flow-shop-scheduling (CBFSS) problem is introduced (Liu and Kozan 2007b). In a sense, the CBFSS problem is a generalised case of the flow-shop scheduling (FSS) problem with blocking, no-wait, limited-capacity buffer or infinite-capacity buffer. By exploiting the structural properties of the combined inter-machine buffer conditions, an innovative constructive algorithm called the *LK* algorithm is proposed to solve the CBFSS problem (Liu and Kozan 2007b).

This chapter will be devoted to the *non-classical* shop scheduling problems. The rest of this chapter is organised as follows. A literature review about the *blocking flow-shop scheduling* (BFSS), *no-wait flow-shop scheduling* (NWFSS), *limited-buffer flow-shop scheduling* (LBFSS) and *blocking job-shop scheduling* (BJSS) problems will be given in the next section. Some known solution techniques proposed for these non-classical FSS problems will be presented in Section 5.3. In Section 5.4, the alternative graph is defined and described, in comparison with the disjunctive graph. In Section 5.5, the topological-sequence algorithm based on an alternative graph model is developed for solving the PFSS, GFSS and JSS problems with blocking. The preeminence of this proposed algorithm is indicated in comparison with two typical algorithms for BPFSS in the literature. In Section 5.6, four different inter-machine buffer conditions are combined to generate a new and more generic non-classical scheduling problem called CBFSS; and an innovative constructive algorithm called the *LK* algorithm is proposed to solve this strongly NP-hard problem. Detailed implementations and sensitivity analysis of the *LK* algorithm are conducted. Finally, some concluding remarks are drawn in the last section.

5.2 Literature Review

5.2.1 Blocking Flow-Shop Scheduling (BFSS)

The classical flow-shop scheduling (FSS) problem is a multi-stage system with unlimited inter-machine buffer storage. In a permutation FSS (PFSS) problem, not only each job has to be processed in the same machine sequence but also the sequence of jobs must be identical on each machine. The general FSS (GFSS) problem may obtain a non-permutation FSS schedule in which the sequence of jobs on different machines may vary.

For the blocking FSS (BFSS) problem, there is no buffer storage between any two successive machines. As a result, a job completed on one machine may block this machine until the next machine is available for processing. A good review for this problem was given by Hall and Sriskandarajah (1996). They proved that the BFSS

problem is NP-hard in the strong sense, for more than two machines. As regards the solution techniques for solving BFSS, McCormick et al. (1989) developed a constructive heuristic known as *Profile Fitting* (PF), which creates a partial sequence by adding the unscheduled job that leads to the minimum sum of blocking times on machines. Abadi et al. (2000) proposed a heuristic based on the idea of increasing the processing times of certain operations in order to establish a connection between no-wait FSS (NWFSS) and blocking FSS (BFSS). This approach, for calculating the value of makespan for a given schedule, has been used by Caraffa et al. (2001) to develop a genetic algorithm to minimise makespan. Recently, Ronconi (2004) developed two hybrid algorithms by employing the *MinMax* (MM) and *Profile Fitting* (PF) algorithms respectively to replace the LPT dispatching rule used in the first step of the *NEH* algorithm of Nawaz et al. (1983). Computation results show that the proposed *MME* and *PFE* hybrid algorithms outperform the *MM*, *PF* and *NEH* algorithms, because the initial solutions used by the *MME* and *PFE* algorithms explore the specific characteristics of blocking conditions while these characteristics can not be explored by the LPT dispatching rule. Furthermore, Ronconi (2005) developed an exact algorithm based on a branch-and-bound scheme, using new lower bounds that exploit the blocking nature and are better than those presented in his earlier paper (Ronconi and Armentano, 2001). Based on a recursive procedure of calculating the makespan (See Section 5.3.2.1), Grabowski and Pempera (2007) developed an efficient tabu search to minimise the makespan of the BFSS problem.

5.2.2 No-Wait Flow-Shop Scheduling (NWFSS)

The *no-wait (or continuous) FSS* (NWFSS) problem occurs when the operations of each job in a flow-shop environment have to be processed continuously from start to end without interruptions. This implies, when necessary, that the start of a job on the first machine may be delayed in order that the operation's completion coincides with the start of the next operation on the subsequent machine. In fact, there are many industries where the NWFSS problem applies. Typical examples include the metal, plastic, chemical, and food industries. For instance, in the food industry, the canning operation must immediately follow the cooking operation to ensure

freshness. Additional typical applications can be found in train scheduling when considering the prioritised trains like express passenger trains, for which the customer has a high cost of waiting during travel.

Therefore, the NWFSS problem has attracted attention from many researchers. Preliminary results about computational complexity for no-wait scheduling problems appeared in Goyal and Sriskandarajah (1988). Hall and Sriskandarajah (1996) also gave a detailed review of the applications and research on this problem and indicated that that the NWFSS problem with the objective of makespan or mean flowtime is NP-complete even for the two-machine case. Rajendran and Chaudhuri (1990) proposed two heuristic algorithms for the *m*-machine NWFSS problem and showed that they are superior to other existing heuristics. Chen *et al.* (1996) later developed a genetic algorithm and compared it with the heuristics of Rajendran and Chaudhuri (1990). Fink and Voβ (2003) considered the application of different kinds of metaheuristics from a practical point of view for the NWFSS problem, and examined the trade-off between running time and solution quality as well as the knowledge and efforts needed to implement and calibrate the algorithms.

5.2.3 Limited-Buffer Flow-Shop Scheduling (LBFSS)

The *limited-buffer flow-shop scheduling* (LBFSS) problem can be considered a generalisation of the classic FSS problem in order to cover a case with limited-capacity inter-machine buffers. Actually, problems of this type create a relatively new direction of research for the practical environments because the limited buffer conditions are often met. However, because of the NP-hardness of this problem, usually the effect of limited buffer storage is ignored, although in industrial production or computer architecture the amount of available buffer storage has a significant influence on the performance of systems. To our knowledge, very few researchers address this problem in the literature. Leisten (1990) presented an overview of how to formulate the flow-shop scheduling problems with limited buffer storage but his proposed heuristics is very simple and can not produce better solutions than those provided by the *NEH* algorithm of Nawaz *et al.* (1983).

Nowicki (1999) dealt with this problem by a sophisticated tabu search approach based on using a non-trivial generalisation of the block elimination.

5.2.4 Blocking Job-Shop Scheduling (BJSS)

As pointed out in the review of scheduling problems with *blocking* and *no-wait* by Hall and Sriskandarajah (1996), the minimisation of the makespan in a no-wait or blocking job shop received surprisingly little attention from either a theoretical or computational perspective.

First, an important issue to be taken into account is the computational complexity of the resulting problem, when studying different versions of the job-shop scheduling (JSS) problem. It is well known that the *classical* JSS problem with unlimited buffers is one of the most computationally intractable NP-hard problems in combinatorial optimisation.

Relevant theoretical results on the *blocking job-shop scheduling* (BJSS) problem were introduced in the control of packet switching communication networks (Arbib *et al.* 1990). A packet switching *k*-network (also called store-and-forward *k*-network, briefly SF k-network) can be represented as an undirected graph $H = (V, E)$, where the vertices stand for processors and the edges for communication links. Each vertex is associated a buffer of capacity *k*. A packet is a message available on a source processor which must be sent to another destination processor. An *acyclic* route for a packet *p* is a directed path on the network associated with *p*, i.e. going from the source to the destination vertex of *p*. By viewing the vertices of a SF 1-network as blocking machines, the problem of finding a sequence of moves such that all packets can reach their respective destinations is therefore a BJSS problem. Arbib *et al.* (1990) also provided a polynomial algorithm for the particular case in which the SF 1-network is a tree.

In the literature, a considerable number of algorithmic improvements have been made for the *classical* JSS problem in the past 30 years. Most of these works are based on the *disjunctive graph* formulation of Roy and Sussman (1964). However, a strong limitation that still remains in the disjunctive graph is that it disregards the

capacity of intermediate buffers between machines. In fact, in many real-life situations especially for train scheduling, the inter-machine buffer capacity has to be seriously taken into account. To incorporate this restriction, the disjunctive graph formulation can be extended to a more general graph model called an *alternative graph*. Mascis and Pacciarelli (2002) studied the *blocking and no-wait job-shop scheduling* (BJSS and NWJSS) problems by means of an alternative graph. They formulated the BJSS and NWJSS problems and investigated the applicability of some effective dispatching rules. These rules include *Avoid Maximum Current Machine* (AMCM), *Select Most Critical Pair* (SMCP), *Select Most Balanced Pair* (SMBP), and *Select Max Sum Pair* (SMSP). Based on these dispatching rules, some heuristic procedures are adapted to solve the BJSS problem.

5.2.5 Remarks

From the literature review, in fact, finding a completely feasible solution for most *non-classical* scheduling problems is not a trivial task. Moreover, searching the optimal solutions for the *non-classical* scheduling problems is much harder than finding the optimums when they are *classical* problems with unlimited buffer. Due to strong NP-hardness for the non-classical scheduling problems, the small computational effort of solution techniques is very valuable in many practical applications. Thus, it is essential to take advantage of the fundamental and specific characteristics of the given problem type. In the next section, some typical constructive algorithms that explore the structural properties of the NWFSS and BFSS problems will be presented in detail.

5.3 Some Typical Solution Techniques

5.3.1 Solution Techniques for NWFSS

The *flow-shop scheduling* (FSS default as PFSS) problem focuses on processing a given set of jobs, where all jobs have to be processed in an identical order on a given number of machines. The *no-wait flow-shop scheduling* (NWFSS) problem has the additional restriction that the processing of each job has to be continuous,

i.e., once the processing of a job begins, there must not be any idle times between the processing of any consecutive operations of this job. If a job does not have to be processed on a machine (zero processing time on this machine), passing could occur without violating continuous processing.

In the literature, the NWFSS problem is generally solved by searching the best *permutation* FSS schedule, for which the permutation of jobs is identical on each machine.

5.3.1.1 Modelled as TSP

In fact, the NWFSS problem with minimising the makespan can be formulated as a *travelling salesman problem* (TSP) with a little modification (Reddi and Ramamoorthy, 1972; and Wismer, 1972).

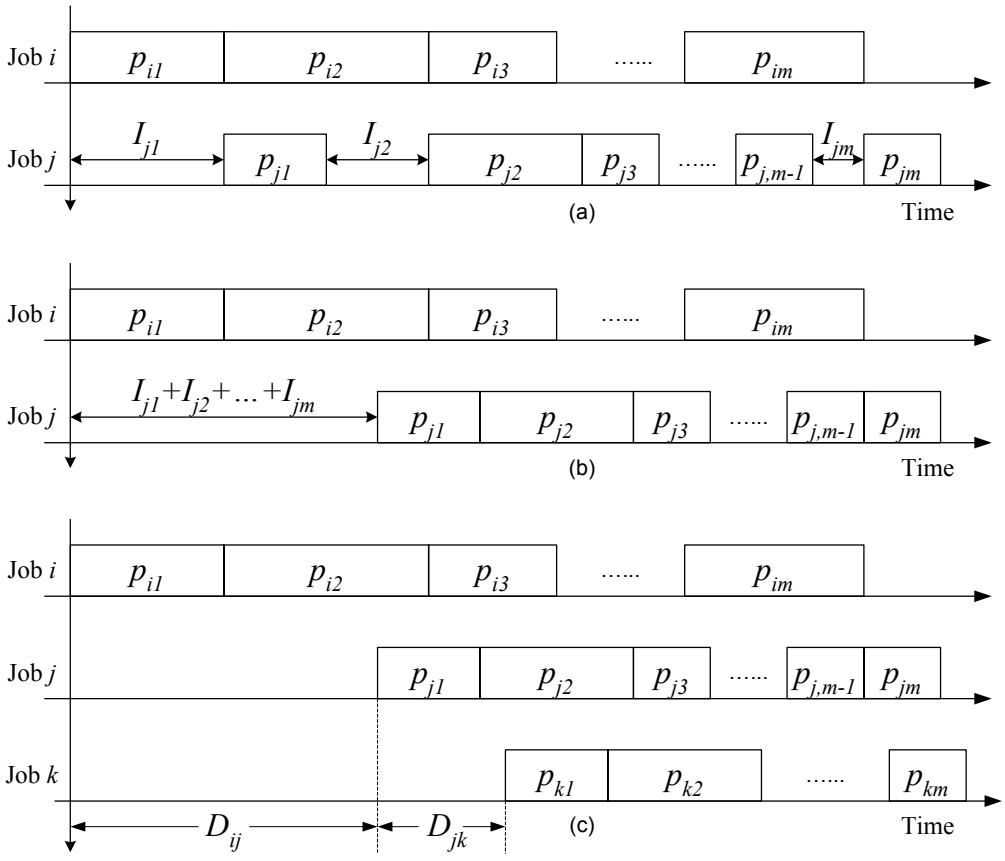

Figure 5-1: The timing of operations on successive jobs. Note that the arrangement in (b) and (c) avoids any interruption during processing all the given jobs.

Suppose that job *i* and job *j* are adjacent in sequence and that job *i* precedes job *j,* as shown in Figure 5-1. In Figure 5-1 (a), I_{jh} denotes the idle time or delay incurred by job *j* on machine *h*; and p_{jh} denotes the processing time of an operation that belongs to job *j* and is processed on machine *h*. To process job *j* without any interruption, the total idle times can be assumed to be aggregated at the start on the first machine, as shown in Figure 5-1(b).

Let D_{ij} denote the total delay incurred by job *j* when it follows job *i* in sequence. That is,

$$D_{ij} = \sum_{h=1}^{m} I_{jh} \tag{5.1}$$

If one schedule consists of three jobs that are processed continuously without any interruption, as illustrated in Figure 5-1(c), then an expression for the makespan of the schedule associated with the sequence *i-j-k* is given by

$$C_{\max} = D_{ij} + D_{jk} + \sum_{h=1}^{m} p_{kh} \tag{5.2}$$

The structure of this expression closely resembles the criterion in the travelling salesman problem (TSP). By the above analysis, the NWFSS problem with the objective of minimising the makespan can be formulated as an asymmetrical *travelling salesman problem* (TSP). In the TSP formulation, each city corresponds to a job, and the intercity distances correspond to the delay pairs D_{ij}. In addition, one dummy city must be added to the problem for representing an idle state, to which the distance from city *j* is $\sum_{h=1}^{m} p_{jh}$, and from which the distance to all other cities is zero. The matrix of this asymmetrical TSP is given in Table 5-1.

Table 5-1: TSP formulation for NWFSS with minimising makespan

−	D_{12}	D_{13}	…	…	D_{1n}	$\sum_{h=1}^{m} p_{1h}$
D_{21}	−	D_{23}	…	…	D_{2n}	$\sum_{h=1}^{m} p_{2h}$
…						…
…						…
…						…
D_{n1}	D_{n2}	D_{n3}	…	…	−	$\sum_{h=1}^{m} p_{nh}$
0	0	0	…	…	0	−

Solving the NWFSS problem aims to construct a permutation $\pi =< \pi_1,...,\pi_n >$ of jobs (π_i denotes the job that is positioned at the i^{th} position of a permutation) that minimise the given objective function. Therefore, the makespan of a NWFSS schedule can be calculated by

$$C_{\max}(\pi) = \sum_{i=2}^{n} D_{\pi_{i-1},\pi_i} + \sum_{h=1}^{m} p_{\pi_n,h} \tag{5.3}$$

This makespan is the sum of two quantities: a sum of sequence-dependent delays on the first machine; and the total processing time of the last job in sequence.

Similarly, the *two-machine* BFSS problem can also be modelled as a special case of the travelling salesman problem, which can be solved in polynomial time by the algorithm of Gilmore and Gomory (1964).

As defined in Chapter 3, the *ready time* (r_j) is the point at which job j is available for processing; the *completion time* (C_j) is the point at which the processing of job j is finished; and the *flowtime* (F_j) is the amount of time job j spends in the system: $F_j = C_j - r_j$. In the same spirit, as the NWFSS problem with minimising *makespan* ($C_{\max} = \max_{1 \le j \le n} C_j$) leads to an asymmetrical travelling salesman problem, the NWFSS problem with minimising *total flowtime* ($F = \sum_{j=1}^{n} F_j$) corresponds to a variation of TSP with cumulative costs, which is also called the *delivery man problem* (Bianco *et al.*, 1993). It is assumed that $r_j = 0$ for all jobs. In this case, continuous processing of a job generally determines an inevitable delay D_{ij} $(1 \le i \le n, 1 \le j \le n, i \ne j)$ on the first machine between the start point of job i and the start point of job j when job j immediately follows job i. The delay may be calculated as

$$D_{ij} = \max_{1 \le k \le m} \left\{ \sum_{h=1}^{k} p_{ih} - \sum_{h=2}^{k} p_{j,h-1} \right\} \tag{5.4}$$

Thus, given a permutation FSS schedule ($\pi =< \pi_1,...,\pi_n >$) with no-wait constraints, the total flowtime can be computed by

$$F(\pi) = \sum_{i=2}^{n} (n+1-i) D_{\pi_{i-1},\pi_i} + \sum_{i=1}^{n} \sum_{h=1}^{m} p_{ih} \qquad (5.5)$$

A numerical example is given in Appendix 1 for illustration. Based on the above formulations, two kinds of constructive heuristics are applied to find approximate (or near-optimal) solutions of the NWFSS problem.

5.3.1.2 Nearest Neighbour Heuristic (NNH)

Constructive heuristic methods build a feasible solution by sequentially completing a (partial) solution according to a particular rule. For example, priority dispatching rules are the simplest constructive methods which successively build a complete sequence by adding jobs according to some preference selection. When the solution quality obtained is satisfactory, such methods may be attractive from a practical point of view, since most basic constructive methods are rather easy to understand and implement.

In the application to solve the NWFSS problem, the *nearest neighbour heuristic* (NNH) algorithm appends at each step a not-yet-included job with a minimal inevitable delay to the last job of the not-yet-complete sequence.

The steps of the NNH algorithm are described in the following.

Step 1: Build up a partial solution M and assume i ($i \in M$) is the last job in M.

Step 2: In NWFSS, continuous processing of a job generally determines an inevitable delay D_{ij} $(1 \le i \le n, 1 \le j \le n, i \ne j)$ on the first machine between the start point of job i and the start point of job j when job j is processed directly after job i. The delay is calculated as

$$D_{ij} = \max_{1 \le k \le m} \left\{ \sum_{h=1}^{k} p_{ih} - \sum_{h=2}^{k} p_{j,h-1} \right\} \quad (\forall j \in J - M)$$

where J is the set of total jobs, job i is defined as the last job in M, and j is a not-yet-included job that may be appended.

Step 3: Let j^* be the most promising job to be selected, if $D_{ij^*} = \min_{\forall j \in J-M} D_{ij}$.

Step 4: Update the partial solution, that is, $i = j^*$ and $M = M \bigcup j^*$. If M is complete, stop; otherwise, go to Step 2 for the next iteration.

However, this strategy seems more reasonable for the reactive scheduling problem at hand, since the jobs positioned first have a significant impact on the objective function.

5.3.1.3 Best Insertion Heuristic (BIH)

Similar to the *NNH* algorithm, the *best insertion heuristic* (BIH) algorithm considers all possible insertions of all not-yet-included jobs while successively building a complete sequence. That is, starting with one initial job, at each step $k = 2,...,n$, all possible combinations of $n-k+1$ jobs with k insertion positions are investigated.

The detailed procedure of the BIH algorithm for NWFSS is given as follows.

Step 1: Build up a partial solution M with only one initial job; and set $k = 2$.

Step 2: At k ($k = 2,...,n$), consider all possible combinations of $n-k+1$ jobs with k insertion positions; and then calculate the objective function values of the resulting new partial solutions ($M' =< \pi_1,...,\pi_k >$) according to the below equations.

If the objective is to minimise the flow time, the equation is:

$$F(M') = \sum_{i=2}^{n}(n+1-i)D_{\pi_{i-1},\pi_i} + \sum_{i=1}^{n}\sum_{h=1}^{m} p_{ih} .$$

If the objective is to minimise the makespan, the equation is:

$$C_{\max}(M') = \sum_{i=2}^{n} D_{\pi_{i-1},\pi_i} + \sum_{h=1}^{m} p_{\pi_n,h} .$$

Step 3: Select M'^*, if its objective function value is minimum.

Step 4: Update the partial solution, i.e. $M = M'^*$ and $k = k+1$. If M is complete or k is greater than n, stop; otherwise, go to Step 2 for the next iteration.

In general, the effectiveness of the constructive methods such as the *nearest neighbour heuristic* (NNH) or the *best insertion heuristic* (BIH) algorithm depend on the initial job. Hence, to explore broader solution space, it would be necessary

to repeat the constructive heuristic for all possible jobs used as the initial job and eventually select the best final sequence.

5.3.2 Solution Techniques for BFSS

Notations:

$J = \{1, 2, \ldots, j, \ldots, n\}$	a set of n jobs
$M = \{1, 2, \ldots, k, \ldots, m\}$	a set of m machines; in a flow shop, each of n jobs has to be processed on machines in the unidirectional order.
O_{jk}	the operation corresponding to the processing of job j on machine k.
p_{jk}	the processing time of operation O_{jk}.
C_{jk}	the completion time of O_{jk}.
B_{jk}	the blocking period of O_{jk}.
D_{jk}	the departure time of O_{jk}, $D_{jk} = C_{jk} + B_{jk}$
D_{j0}	the ready time of job j, i.e. the moment when job j can begin its processing on the first machine.

Based on the above notations, the *blocking flow-shop scheduling* (BFSS) problem can be stated as follows.

Since the blocking flow shop has no intermediate buffers between any two successive machines, a job j $(\forall j = 1, 2, \ldots, n)$, having completed processing of its operation O_{jk}, cannot leave the machine k $(k = 1, 2, \ldots, m-1)$ until the next machine $k+1$ is available. If the machine $k+1$ is not available, the job j has to block machine k. The objective of the BFSS problem is to find a permutation FSS schedule such that all blocking conditions are satisfied and the makespan is minimal.

Thus, each BFSS schedule can be represented by the permutation $\pi =< \pi_1,\ldots,\pi_n >$ on the job set J. Let Π denote the set of all feasible permutations. The objective is to find such a permutation $\pi^* \in \Pi$, that

$$C_{\max}(\pi^*) = \min_{\pi\in\Pi} C_{\max}(\pi) \tag{5.6}$$

where $C_{\max}(\pi)$ is the time required to complete all jobs on machines in the processing order given by π. It is obvious that, for a given permutation schedule of BFSS, the makespan is equal to the departure time of the last job on the last machine, namely, $C_{\max}(\pi) = D_{\pi_n,m}$.

5.3.2.1 Recursive Procedure

Give a permutation schedule of BFSS, the departure times of each job on each machine can be recursively computed according to the following expressions (Ronconi, 2004).

$$D_{\pi_1,0} = 0$$

$$D_{\pi_1,k} = \sum_{l=1}^{k} p_{\pi_1,l},\ \forall k = 1,\ldots,m-1$$

$$D_{\pi_j,0} = D_{\pi_{j-1},1},\ \forall j = 2,\ldots,n$$

$$D_{\pi_j,k} = \max\{D_{\pi_j,k-1} + p_{\pi_j,k}, D_{\pi_{j-1},k+1}\},\ \forall j = 2,\ldots,n; k = 1,\ldots,m-1$$

$$B_{\pi_j,k} = \begin{cases} D_{\pi_{j-1},k+1} - (D_{\pi_j,k-1} + p_{\pi_j,k}) & \text{if } D_{\pi_{j-1},k+1} - (D_{\pi_j,k-1} + p_{\pi_j,k}) > 0 \\ 0 & \text{otherwise} \end{cases},$$

$$\forall j = 2,\ldots,n; k = 1,\ldots,m-1$$

$$D_{\pi_j,m} = D_{\pi_j,m-1} + p_{\pi_j,m},\ \forall j = 1,\ldots,n \tag{5.7}$$

where $D_{\pi_j,0}$ is the ready time of job π_j on the first machine. In the above recursion procedure, the departure times of the first job on every machine are calculated first, then the second job, and so on until the last job. The makespan of a BFSS schedule is obtained by the calculation of $D_{\pi_n,m}$. A numerical example is given in Appendix 2 for illustrating the recursive procedure.

5.3.2.2 Directed Graph

For the *blocking flow-shop scheduling* (BFSS) problem, in addition to recursive procedure, another way of obtaining the makespan is through a so-called *directed graph* (Ronconi, 2004).

In the directed graph, node (π_j, k) denotes the departure time ($D_{\pi_j,k}$) of job π_j ($\forall j = 1,\ldots n-1$) on machine k ($\forall k = 1,\ldots m-1$). This node has two outgoing arcs: one arc is directed to node $(\pi_j, k+1)$ and has a weight equal to $p_{\pi_j,k+1}$, while the other arc goes to $(\pi_{j+1}, k-1)$ with a zero weight for representing the blocking condition. Exceptionally, nodes $(\pi_j, 0)$, (π_j, m) and (π_n, k) have only one outgoing arc connected to nodes $(\pi_j, 1)$, $(\pi_{j+1}, m-1)$ and $(\pi_n, k+1)$ with weights $p_{\pi_j,1}$, zero and $p_{\pi_n,k+1}$, respectively. Note that (π_n, m) has no outgoing arcs and its departure time is equal to the makespan, i.e. $C_{\max}(\pi) = D_{\pi_n,m}$.

For illustration, a directed graph for a 5-job 4-machine ($n = 5, m = 4$) BFSS instance is drawn in Figure 5-2, with the given permutation schedule, $\pi =< \pi_1,\ldots,\pi_5 >=< 1,2,3,4,5 >$.

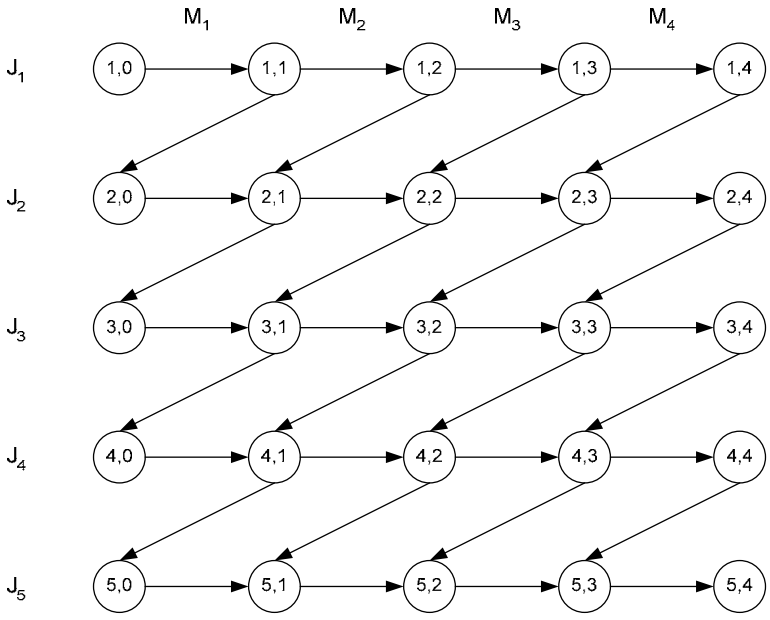

Figure 5-2: Directed Graph for a 5-job 4-machine BFSS instance

The completion time of a job π_j is given by the longest path between nodes $(\pi_1, 0)$ and (π_j, m). Therefore, the makespan is obtained by the longest path between nodes $(\pi_1, 0)$ and (π_n, m), known as the *critical path*.

Formula I

Given a machine k, consider the longest path passing through machine k. Its length is calculated by the following formula:

$$PL(k) = \sum_{q=1}^{k} p_{\pi_1,q} + \sum_{r=2}^{n} \max\{p_{\pi_r,k}, p_{\pi_{r-1},k+1}\} + \sum_{q=k+1}^{m} p_{\pi_n,q} \tag{5.8}$$

where $p_{\pi_j,m+1} = 0$ for every $j = 1,\ldots,n$.

Using the above example, $PL(3)$, the length of the longest path passing through machine 3, is calculated by Eq. (5.8):

$$\begin{aligned} PL(3) = {} & (p_{11} + p_{12} + p_{13}) \\ & + (\max\{p_{2,3}, p_{1,4}\} + \max\{p_{3,3}, p_{2,4}\} + \max\{p_{4,3}, p_{3,4}\} + \max\{p_{5,3}, p_{4,4}\}) \\ & + (p_{54}) \end{aligned}$$

In the calculation of $PL(k)$ ($k = 3$), the processing times p_{11}, p_{12} and p_{13} of the first job compose the first term. The second term is comprised of the sum of maximum values of consecutive jobs on machine 3 by satisfying the blocking conditions. For example, to reach node $(3,3)$ from node $(2,3)$, the longest processing time between $p_{2,4}$ and $p_{3,3}$ is selected. The last term represents the path to reach the last node $(\pi_n, m) = (5,4)$ from node $(\pi_n, k) = (5,3)$.

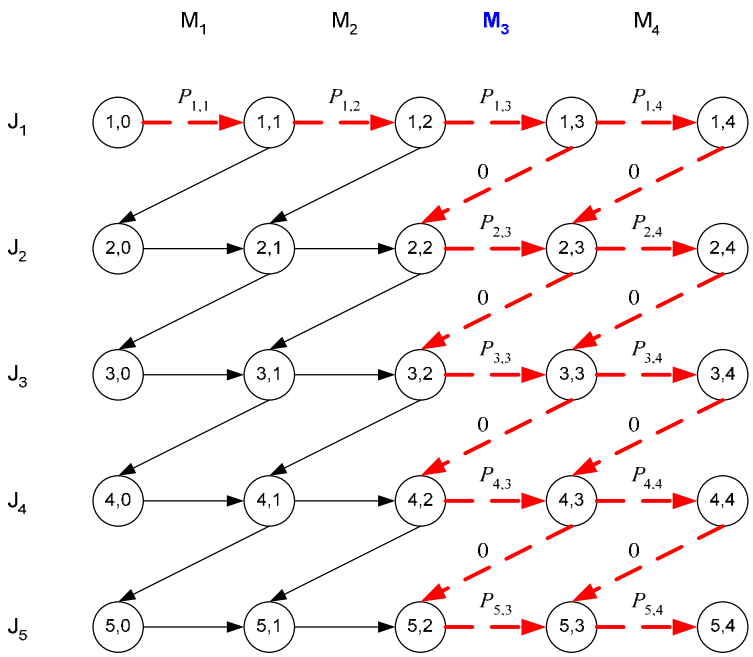

Figure 5-3: Directed Graph for highlighting the longest path through machine 3

This longest path through machine 3 is highlighted by the red thicker lines, as shown in Figure 5-3.

Formula II

It is observed that the critical path does not necessarily pass by all nodes related to a certain machine. Given an operation $O_{\pi_j,k}$ that belongs to job π_j and is processed on machine k, the length of the longest path passing through $O_{\pi_j,k}$ is computed by the following expression:

$$\begin{aligned} PL(\pi_j,k) = &\sum_{q=1}^{k} p_{\pi_1,q} + \sum_{r=2}^{j} \max\{p_{\pi_r,k}, p_{\pi_{r-1},k+1}\} \\ &+ p_{\pi_j,k+1} + \sum_{r=j+1}^{n} \max\{p_{\pi_r,k+1}, p_{\pi_{r-1},k+2}\} \\ &+ \sum_{q=k+2}^{m} p_{\pi_n,q} \end{aligned} \tag{5.9}$$

where $p_{\pi_j,m+1} = p_{\pi_j,m+2} = 0$ for every $j = 1,\ldots,n$. Figure 5-4 illustrate the longest path through the operation O_{42} corresponding to the job $\pi_j = \pi_4 = 4$ and the machine $k = 2$.

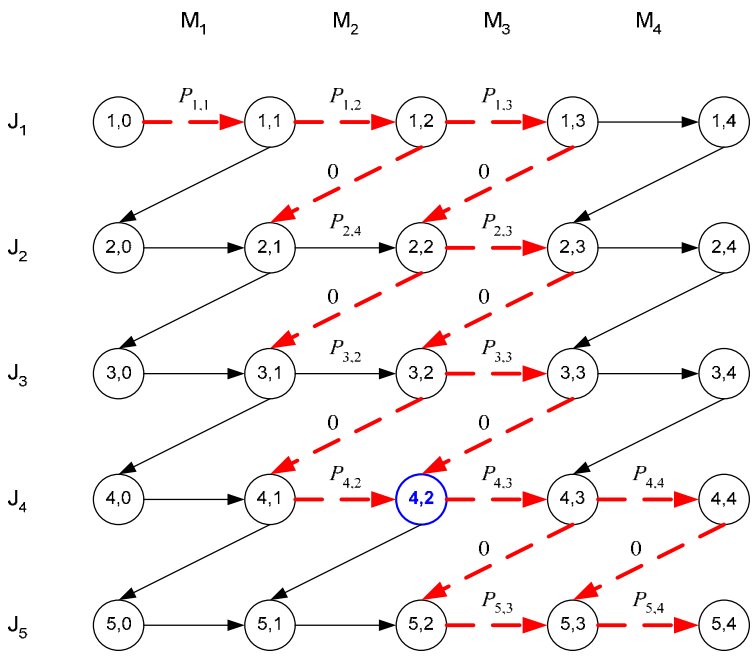

Figure 5-4: Directed Graph for highlighting the longest path through node (4,2)

According to Figure 5-4, the length of the longest path $PL(4,2)$ is computed as follows:

$$PL(4,2) = (p_{11} + p_{12}) + (\max\{p_{2,2}, p_{1,3}\} + \max\{p_{3,2}, p_{2,3}\} + \max\{p_{4,2}, p_{3,3}\}) + (p_{4,3}) + (\max\{p_{5,3}, p_{4,4}\}) + (p_{5,4})$$

By analysing the directed graph, it is indicated that any path connecting the node $(\pi_1, 0)$ and (π_n, m) will definitely pass by the arc associated with $p_{\pi_1,1}$ (i.e. processing time of the first job on the first machine) and the arc associated with $p_{\pi_n,m}$ (i.e. processing time of the last job on the last machine). It is also noted that we can generate several lower bounds on the makespan, formed by various paths that pass through different machines or different nodes.

5.3.2.3 MinMax (MM) Heuristic

An effective constructive heuristic called a *MinMax* (MM) algorithm is proposed for the BFSS problem (Ronconi,2004). The *MM* heuristic considers the characteristics of BFSS in the construction of the solution. Initially, inspired by the well-known Johnson's rule, on the first and last position, the *MM* algorithm sets the jobs whose processing times are the shortest on the first and last machine,

respectively. Then, from the second position, the *MM* algorithm adds the job of the next position in the sequence by minimising the length of the critical path. In order to decrease the sums by Eq.(5.8) or (5.9), the *MM* algorithm selects the next job that leads to the smallest value of the following expression:

$$\alpha\sum_{k=1}^{m-1}|\,p_{v,k}-p_{u,k+1}\,|+(1-\alpha)\sum_{k=1}^{m}p_{v,k} \tag{5.10}$$

where p_{jk} represents the processing time of job j on machine k, and parameter α ($0<\alpha<1$) is used to ponder those terms. This strategy of selecting the next job is based on the following lemma (Ronconi and Armentano, 2001).

***Lemma* I**

Let $A=a_1,a_2,\ldots,a_t$ and $B=b_1,b_2,\ldots,b_t$ be two sets of sequenced positive integer numbers and define $S(A,B)=\sum_{i=1}^{t}\max(a_i,b_i)$. Thus, $S(A,B)$ is minimised if A and B are sequenced in such a way that for each pair (a_i,b_i), there is $\forall k\in\{1,2,\ldots,t\}$, so that a_i and b_i correspond to the k^{th} smallest element of A and B, respectively.

Since it is very hard to order the processing times of the jobs according to the above lemma, the selection strategy aims to construct a solution where the difference between them is mostly reduced. In order to reach this goal, the first term of Eq. (5.10) is the modulus of the differences between processing times of consecutive jobs on consecutive machines. The second term composes this expression to prioritise jobs with the smallest processing times. Since the critical path will also be composed of the sum of m processing times, it is relevant to increase the probability of jobs with the smallest processing times to compose this path, by positioning them in the beginning of the sequence.

A brief description of the *MM* heuristic steps is presented below.

Step 1: On the first and the last position, set those jobs with the shortest processing times on the first and the last machines, respectively. Make the next position, *next_pos* = 2.

Step 2: From the non-set jobs, select the one that leads to the smallest value of Eq. (5.10). Set this job on position *next_pos*.

Step 3: Make *next_pos* = *next_pos* + 1. If *next_pos* is equal to n, stop. Otherwise, go back to Step 2.

5.3.2.4 NEH Algorithm

The *NEH* algorithm (**N**awaz, **E**nscore and **H**am, 1983) appears to be the best polynomial heuristic algorithm to minimise the makespan of the *classical* FSS problem with unlimited buffer.

The description of the *NEH* procedure has been given in Section 4.3.1. Initially, this algorithm generates a list of priorities among jobs, using the LPT dispatching rule. The two jobs with the highest priority are selected from the list and the two possible partial sequences for these two jobs are generated. These two sequences are evaluated and the best partial sequence with the minimum makespan is chosen. It is noted that the relative positions of those two jobs do not change during the whole algorithm procedure. Then, the job with the third highest priority is selected and three sequences are generated by inserting this job in the beginning, middle and end of the sequence obtained in the previous step. After evaluation, the best partial sequence found will set the relative positions of these three jobs. This process is repeated until all jobs are determined and the final complete sequence is obtained.

5.3.2.5 MME Algorithm

Ronconi (2004) proposed a so-called *MME* algorithm to minimise the makespan of the BFSS problem, by implementing the *MM* heuristic instead of the LPT dispatching rule as a starting point used in the NEH algorithm. The proposed algorithm is a two-phase heuristic, henceforth called the ***MME*** algorithm.

A brief description of the *MME* procedure is presented below.

Step 1: Order the n jobs by implementing the *MinMax* (MM) heuristic.

Step 2: Take the first two jobs and schedule them in order to minimise the partial makespan as if there were only two jobs.

Step 3: For $k \leftarrow 3$ to $k \leftarrow n$, do:
Insert the k^{th} job at the each k possible position, which minimises the partial makespan.

Since the *MM* heuristic depends on the parameter α, tests need to be conducted again using this parameter to select the best version of the *MME* heuristic.

5.3.2.6 Profile Fitting (PF) Algorithm

Compared to the LPT dispatching rule and *MM* heuristic, the *profile fitting* (*PF*) algorithm proposed by (McCormick *et al.,* 1989) can also construct an ordered list of priorities among jobs for providing the initial step in the *NEH* algorithm procedure.

Initially, the *PF* algorithm selects the first job that has the smallest sum of processing times for the first position. For the second position, in order to determine the job that will take the next position in the sequence, each non-sequenced job is examined. The examination procedure at each position is given as follows.

For each candidate job v, the departure times D_{vk} on each machine k are calculated, as if this job were in the position for which it is a candidate to be inserted after its predecessor, a given job u. With these values, the blocking time (i.e. B_{vk}) of job v on machine k can be calculated by the below expression:

$$B_{vk} = D_{vk} - p_{vk} - D_{uk} \tag{5.11}$$

This measure is calculated on each machine for each candidate job. The candidate job that has the smallest sum of those measures is selected as the next job. This way of selecting the next job is repeated until all the jobs are sequenced.

The detailed procedure of the *PF* algorithm is given as follows.

Step 1: Determine the first job that has the smallest sum of processing times. The given job (job u) is set as the first job.

Step 2: For each non-selected job (job v), as if job v is inserted right after job u, calculate the sum of blocking times of job v on all machines, i.e. $\sum_{k=1}^{m} B_{vk} = \sum_{k=1}^{m} (D_{vk} - p_{vk} - D_{uk})$. Then, select the best candidate job v^* that leads to the minimum value, i.e.

$$v^* = \arg\min_{v \in J'} (\sum_{k=1}^{m} B_{vk})$$

where J' is the set of non-selected jobs. The given job (job u) is reset by the selected job v^*.

Step 3: If all the jobs are determined, then stop; otherwise, go to Step 2.

5.3.2.7 PFE Algorithm

Similar to the *MME* algorithm, Ronconi (2004) proposed a so-called *PFE* algorithm, by implementing the *PF* algorithm instead of LPT dispatching rule to generate the initial list of priorities among jobs used in the first step of the *NEH* algorithm. The proposed algorithm is also a two-phase hybrid heuristic, henceforth called the *PFE* algorithm.

A brief description of the *PFE* procedure is presented below.

Step 1: Order the *n* jobs by implementing the *PF* heuristic.

Step 2: Take the first two jobs and schedule them in order to minimise the partial makespan as if there were only two jobs.

Step 3: For $k \leftarrow 3$ to $k \leftarrow n$, do:
Insert the k^{th} job at the each k possible position, which minimises the partial makespan.

The *NEH* algorithm is strongly influenced by the initial job ordering. In comparison to the LPT dispatching rule, the initial solution used by *PFE* and *MME* algorithms exploits specific characteristics of blocking conditions. These characteristics are not explored by the LPT dispatching rule. Therefore, as two-

phase hybrid heuristics, the *PFE* and *MME* algorithms certainly outperform the *NEH*, *PF* or *MM* algorithm, for minimising the makespan of the BFSS problem.

5.4 Alternative Graph Model

5.4.1 Limitation of Disjunctive Graph

From the point of view of the modelling techniques, most research works in scheduling are based on the disjunctive graph formulation of Roy and Sussmann (1964). The disjunctive graph model has been extensively studied in order to develop efficient solution techniques for solving theoretic scheduling problems. However, a strong limitation that still remains in these models is that they disregard the capacity of intermediate buffers. In fact, in many real-life situations buffer capacity should be taken into account. In scheduling theory, the absence of a buffer is often modelled with *blocking* or *no-wait* constraints. In particular, a *blocking* constraint models the absence of storage capacity between machines, while a *no-wait* constraint occurs when two consecutive operations in a job must be processed without any interruption. To incorporate this restriction, the disjunctive graph formulation can be extended to be a more general graph model called an *alternative graph*.

5.4.2 Definition of Alternative Graph

In the alternative graph model, we focus on the sequencing of operations rather than jobs. There are a set of N operations $\mathrm{O} = \{o_0, o_1, \ldots, o_N\}$ to be performed on a set of m machines. Each operation o_i requires a specified amount of processing time p_i on a given machine $M(i)$, and cannot be interrupted from its starting time e_i to its completion time $c_i = e_i + p_i$. Operations o_0 and o_N, respectively called *First* and *Last*, are dummy operations with zero processing time. Each machine can process only one operation at a time.

There is a set of precedence relationships among operations. A precedence relation $(o_i \to o_j)$ is a constraint on the starting time of operation o_j. More precisely, in scheduling problems considering blocking constraints, the starting time of the successor o_j must be greater or equal to the starting time of the predecessor o_i plus a given delay f_{ij}. In the alternative graph model, the delay f_{ij} can be either positive, null or negative. A positive delay f_{ij} may represent the fact that operation o_j may start processing after the completion of operation o_i, plus a possible setup or storage time. A delay smaller or equal to zero may represent synchronization between the starting times of two operations. Finally, for dummy *first* and *last* operations, we define that o_0 precedes $o_1,\ldots,o_N$ and o_n follows $o_0,o_1,\ldots,o_{N-1}$.

In the alternative graph model, precedence relations are divided into two sets: ***fixed*** arcs and ***alternative*** arcs instead of ***disjunctive*** arcs. Thus, a schedule is considered as an assignment of starting times $e_0,e_1,\ldots,e_N$ of operations $o_0,o_1,\ldots,o_N$, respectively, such that one precedence relation for each pair of the alternative arcs is selected. We assume the starting time of o_0 is zero, namely, $e_0 = 0$. The objective is to minimise e_N, the starting time of o_N.

Associating a node to each operation, the no-wait and blocking scheduling problem can be represented by the triple $G = (\mathrm{O}, C, A)$, called the *alternative graph*. In the alternative graph, there is a set of nodes O, a set of fixed-direction conjunctive arcs C and a set of alternative arcs A. If $((o_i \to o_j),(o_h \to o_k)) \in A$, we say that $(o_i \to o_j)$ is the alternative of $(o_h \to o_k)$ or $(o_h \to o_k)$ is the alternative of $(o_i \to o_j)$. In this definition, an alternative arc belongs to one pair only. Given a pair of alterative arcs $((o_i \to o_j),(o_h \to o_k)) \in A$, we say that $(o_i \to o_j)$ is selected whereas $(o_h \to o_k)$ should be forbidden. If neither $(o_i \to o_j)$ nor $(o_h \to o_k)$ is selected, we say that this pair of alternative arcs is unselected. The arc lengths can be positive, null or negative. The solution is feasible if there are no positive length cycles in the resulting graph.

5.4.3 Difference between Disjunctive Graph and Alternative Graph

Let us first consider a pair of consecutive operations in a job, say o_i and $o_{SJ[i]}$ (i.e. the successor of o_i in the same job), and assume that $o_{SJ[i]}$ must start processing within k $(k \geq 0)$ time units after the completion of o_i. This is called the *perishability* constraint, which can be formulated as

$$e_i + p_i \leq e_{SJ[i]} \leq e_i + p_i + k$$

$$e_{SJ[i]} - e_i \geq p_i$$

$$e_i - e_{SJ[i]} \geq -p_i - k$$

$$e_i + p_i = e_{SJ[i]} \text{ if } k = 0,$$

since it represents, for example, the fact that a job deteriorates when stored for more than k time units between the two consecutive operations.

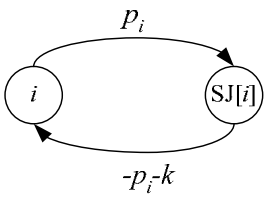

Figure 5-5: Perishability Constraint represented by a pair of alternative arcs

In terms of the definition of alternative arcs, we can represent this *perishability* constraint with a pair of alternative arcs $((o_i \rightarrow o_{SJ[i]}),(o_{SJ[i]} \rightarrow o_i))$ illustrated in Figure 5-5, with lengths $f_{i,SJ[i]} = p_i$ and $f_{SJ[i],i} = -p_i - k$, respectively. If $k = 0$, we have a tight no-wait constraint.

Let us now consider the blocking constraints, which are used to model the absence of storage capacity between machines. We distinguish two types of operations, namely ***ideal*** and ***blocking***. We say that an *ideal* operation remains on a machine from its starting time to its completion time, and then at once leave this machine which becomes immediately available for processing other operations. On the contrary, a *blocking* operation may remain on a machine even after its completion time, thus blocking it. More precisely, a blocking operation has the following two characteristics:

(1) Each blocking operation o_i is associated with another operation ($o_{SJ[i]}$) which immediately follows o_i in the same job and will be executed on a different machine.

(2) Operation o_i having completed processing on machine $M(i)$, remains on it until the machine $M(SJ[i])$ becomes available for processing $o_{SJ[i]}$. The machine $M(i)$ is therefore blocked and cannot process any other operation until o_i leaves $M(i)$ at time $e_{SJ[i]}$, the starting time of operation $o_{SJ[i]}$.

Now let us see the difference between the disjunctive graph and the alternative graph. Consider two operations o_i and o_j, where o_i is blocking and $M(i) = M(j)$. In fact, in the disjunctive graph, the pairs of disjunctive arcs are all in the form of $((o_i \to o_j),(o_j \to o_i))$, where o_i and o_j are two *ideal* operations to be processed on the same machine. In a sense, the alternative graph is an extension of the disjunctive graph when dealing with the non-classical scheduling problems and models under specific properties such as blocking or no-wait conditions in the design of efficient solution techniques.

In the alternative graph model, since o_i is blocking, we associate them with alternative arcs instead of disjunctive arcs. Since o_i and o_j cannot be executed at the same time, each directed arc represent the fact that one operation must be processed before the other. If o_i precedes o_j, and since o_i is blocking, machine $M(i)$ can start processing o_j only after the starting time of $o_{SJ[i]}$ when o_i leaves $M(i)$. Hence, we represent this situation by $o_{SJ[i]} \to o_j$ of length $a_{SJ[i]\to j} = 0$ instead of $o_i \to o_j$. Whether replacing arc $o_j \to o_i$ depends on whether o_j is blocking or ideal. If o_j is ideal, then we keep the arc $o_j \to o_i$ whose length is $a_{j\to i} = p_j$. If o_j is blocking, then we replace $o_j \to o_i$ by $o_{SJ[j]} \to o_i$ of length $a_{SJ[j]\to i} = 0$, implying that this operation blocks a machine until the downstream machine becomes available.

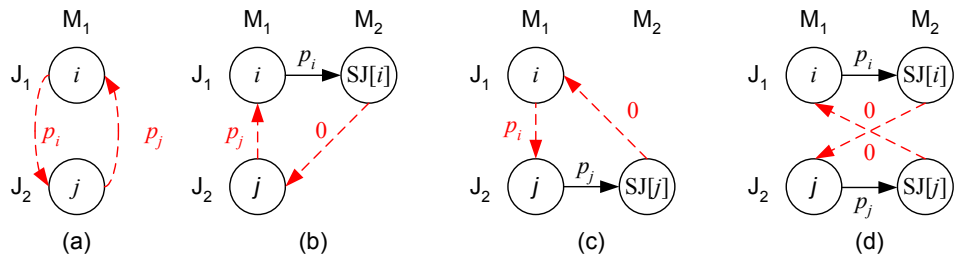

Figure 5-6: (a) The disjunctive arcs $((o_i \to o_j),(o_j \to o_i))$ if both o_i and o_j are ideal; (b) The alternative arcs $((o_{SJ[i]} \to o_j),(o_j \to o_i))$ if o_i is blocking and o_j is ideal; (c) The alternative arcs $((o_{SJ[i]} \to o_j),(o_j \to o_i))$ if o_i is ideal and o_j is blocking; (d) The alternative arcs $((o_{SJ[i]} \to o_j),(o_j \to o_i))$ if both o_i and o_j are blocking

See Figure 5-6 for illustration, where a pair of two alternative arcs (or disjunctive arcs) in each case is highlighted by *red dashed* lines.

In comparison, Figures 5-7(a) and 5-7(b) illustrate how a disjunctive graph transforms to the alternative graph for a three-job four-machine blocking flow-shop scheduling instance.

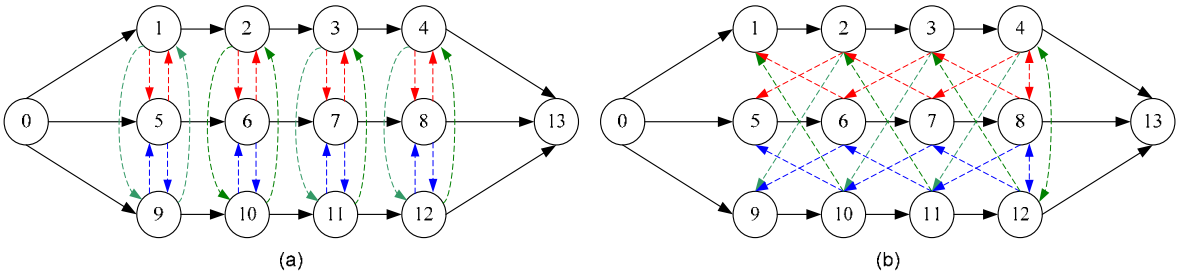

Figure 5-7: (a) The disjunctive graph without considering blocking constraints; (b) Transformation of the disjunctive graph to the alternative graph for considering blocking constraints

In the alternative graph in Figure 5-7(b), for each pair of operations to be executed on the same machine, there is a pair of alternative arcs replacing the pair of disjunctive arcs. For example, in the disjunctive graph in Figure 5-7(a) for operations o_1 and o_5 processed on the same machine, there is a pair of disjunctive arcs $((o_1 \to o_5),(o_5 \to o_1))$. The alterative graph for considering the blocking conditions, this pair of disjunctive arcs $((o_1 \to o_5),(o_5 \to o_1))$ is replaced by a pair of alternative arcs $((o_{SJ[1]} \to o_5),(o_{SJ[5]} \to o_1)) = ((o_2 \to o_5),(o_6 \to o_1))$, where operation

$o_{SJ[i]}$ immediately follows o_i in the same job and will be executed on a different machine. Note that a job, having completed processing on the last machine, leaves the system at once. Hence, the last machine is always available and operations o_4, o_8 and o_{12} are not blocking.

For better understanding, see the difference in the total pairs of undirected arcs (i.e. disjunctive arcs or alternative arcs respectively) between the disjunctive graph and the alternative graph.

For the above example, the total number of alternative (or disjunctive) arcs is 12, as enumerated in Table 5-2.

Table 5-2: The total pairs of undirected arcs in disjunctive graph and alternative graph

Machine Type	Pair of Operations	Disjunctive Arcs	Alternative Arcs
M(1) = M(5) = M(9)	o_1 and o_5	$((o_1 \to o_5),(o_5 \to o_1))$	$((o_2 \to o_5),(o_6 \to o_1))$
	o_1 and o_9	$((o_1 \to o_9),(o_9 \to o_1))$	$((o_2 \to o_9),(o_{10} \to o_1))$
	o_5 and o_9	$((o_5 \to o_9),(o_9 \to o_5))$	$((o_6 \to o_9),(o_{10} \to o_5))$
M(2) = M(6) = M(10)	o_2 and o_6	$((o_2 \to o_6),(o_6 \to o_2))$	$((o_3 \to o_6),(o_7 \to o_2))$
	o_2 and o_{10}	$((o_2 \to o_{10}),(o_{10} \to o_2))$	$((o_3 \to o_{10}),(o_{11} \to o_2))$
	o_6 and o_{10}	$((o_6 \to o_{10}),(o_{10} \to o_6))$	$((o_7 \to o_{10}),(o_{11} \to o_6))$
M(3) = M(7) = M(11)	o_3 and o_7	$((o_3 \to o_7),(o_7 \to o_3))$	$((o_4 \to o_7),(o_8 \to o_3))$
	o_3 and o_{11}	$((o_3 \to o_{11}),(o_{11} \to o_3))$	$((o_4 \to o_{11}),(o_{12} \to o_3))$
	o_7 and o_{11}	$((o_7 \to o_{11}),(o_{11} \to o_7))$	$((o_8 \to o_{11}),(o_{12} \to o_7))$
The last Machine M(4) = M(8) = M(12)	o_4 and o_8	$((o_4 \to o_8),(o_8 \to o_4))$	same
	o_4 and o_{12}	$((o_4 \to o_{12}),(o_{12} \to o_4))$	same
	o_8 and o_{12}	$((o_8 \to o_{12}),(o_{12} \to o_8))$	same

5.4.4 Feasibility Analysis of Alternative Graph

For feasibility analysis, we draw a disjunctive graph for a two-job two-machine job-shop scheduling problem without blocking constraints. In this example, each job consist of two operations, i.e. $J_1 = \{o_1, o_2\}$, $J_2 = \{o_3, o_4\}$. Job J_1 visits machine M_1 then machine M_2. Job J_2 visits machine M_2 then machine M_1. In addition,

there have $M_1 = M(1) = M(3)$ and $M_2 = M(2) = M(4)$, i.e. operations o_1 and o_3 are processed on the same machine M_1; operations o_2 and o_4 are processed on the same machine M_2, as shown in Figure 5-8(a).

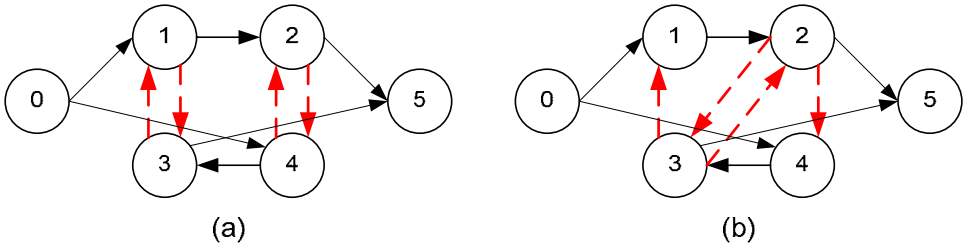

Figure 5-8: (a) Disjunctive graph for a 2-job 2-machine job shop without blocking constraints; (b) Alternative graph for a 2-job 2-machine job shop with blocking constraints.

If considering the blocking constraints, the two pairs of disjunctive arcs $((o_1 \to o_3),(o_3 \to o_1))$ and $((o_2 \to o_4),(o_4 \to o_2))$ are replaced by the two pairs of alternative arcs $((o_2 \to o_3),(o_3 \to o_1))$ and $((o_2 \to o_4),(o_3 \to o_2))$. Thus, the corresponding alternative graph is drawn in 5-8(b).

For this two-job two-machine job-shop scheduling problem with blocking conditions, we can enumerate all of four (feasible or infeasible) schedules by choosing at most one arc from each pair of alternative arcs (only two pairs for this example, $((o_2 \to o_3),(o_3 \to o_1))$ and $((o_2 \to o_4),(o_3 \to o_2))$), respectively illustrated in Figures 5-9, 5-10, 5-11 and 5-12. For simplicity, assume that the processing time for each operation is the same as one time unit.

1) If the alternative arcs are chosen as $(o_3 \to o_1)$ and $(o_2 \to o_4)$, the schedule is infeasible because it is cyclic (i.e. $o_1 \to o_2 \to o_4 \to o_3 \to o_1$), illustrated in Figure 5-9 In this case, the Gantt chart cannot be drawn because the schedule is cyclic or infeasible.

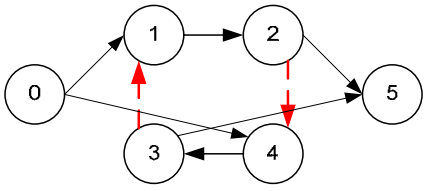

Figure 5-9: One infeasible schedule of a blocking 2-job 2-machine job shop with selected alternative arcs $(o_3 \to o_1)$ and $(o_2 \to o_4)$

2) If the choosing alternative arcs are $(o_3 \to o_1)$ and $(o_3 \to o_2)$, the schedule and its corresponding Gantt chart are presented in Figure 5-10.

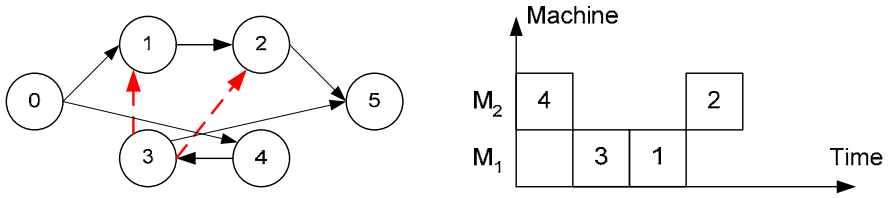

Figure 5-10: One feasible schedule of a blocking 2-job 2-machine job shop with selected alternative arcs $(o_3 \to o_1)$ and $(o_3 \to o_2)$

3) If the choosing alternative arcs are $(o_2 \to o_3)$ and $(o_2 \to o_4)$, this schedule is shown in Figure 5-11.

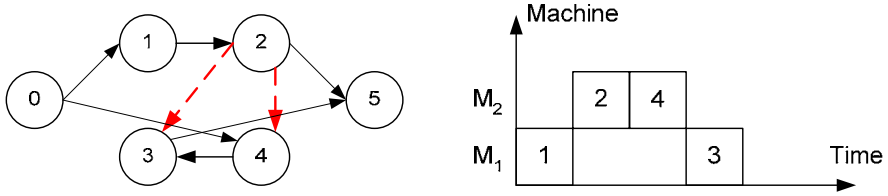

Figure 5-11: One feasible schedule of a blocking 2-job 2-machine job shop with selected alternative arcs $(o_2 \to o_3)$ and $(o_2 \to o_4)$

4) If the choosing alternative arcs are $(o_2 \to o_3)$ and $(o_3 \to o_2)$, whether the schedule is feasible depends on the particular context, described in Figure 5-12 and analysed in the following.

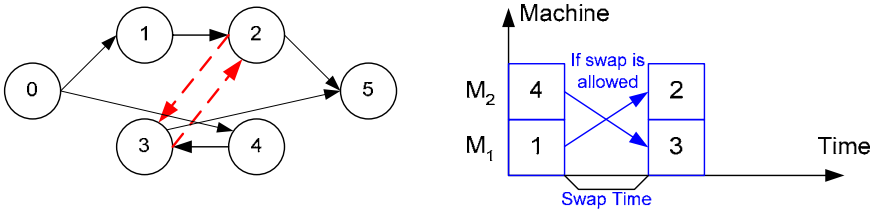

Figure 5-12: One (feasible or infeasible) schedule of a blocking 2-job 2-machine job shop with selected alternative arcs $(o_2 \to o_3)$ and $(o_3 \to o_2)$; note that the schedule is infeasible if swap is not allowed and the Gantt Chart can be drawn only for swap-allowed blocking case.

This blocking situation shown in Figure 5-12 is called "***deadlock***" as well. In such a deadlock situation, we need to distinguish two cases of blocking: *swap-allowed blocking* and *no-swap blocking*. In this case, a cycle $(o_2 \to o_3 \to o_2)$ or

$(M(2) \to M(3) \to M(2))$ may occur when selecting $(o_2 \to o_3)$ and $(o_3 \to o_2)$. In such a blocking situation, all jobs in the cycle must move simultaneously to the next machine in a cycle of machines. For this example, the operation o_2 of job $J_1 = \{o_1, o_2\}$ can be processed on $M_2 = M(2)$ before machine $M_2 = M(4)$ can be released by job $J_2 = \{o_3, o_4\}$ when o_3 can start to be processed on $M_1 = M(3)$ .. However, the operation o_3 of job $J_2 = \{o_3, o_4\}$ can be processed on $M_1 = M(3)$ before machine $M_1 = M(1)$ can be released by job $J_1 = \{o_1, o_2\}$ when o_2 can start to be processed on $M_2 = M(2)$. To be feasible, a "***swap***" manipulation is needed because each job is waiting for a machine that is blocked by another job in the cycle.

It is intuitive that, depending on the particular context, a swap may be allowed or not. The deadlock situation is similar to a conflict between one outbound train and one inbound train in a railway track. Thus, the deadlock situation should be strictly prohibited in the train scheduling problems, namely, the train schedule should be deadlock-free. The deadlock situation will be further discussed in the next chapter on train scheduling.

The swap is allowed when the jobs can move independently of each other. On the contrary, the swap is not allowed when the jobs can move strictly after the subsequent resource becomes available. In this case, we say that the operation is a ***no-swap-blocking operation***. A cycle of blocking operations, in which at least one operation is no-swap, is infeasible. In the train scheduling problems, for example, all the operations are *no-swap-blocking* operations. Therefore, when modelling a shop scheduling problem with blocking constraints, two different situations are classified, namely, *no-swap blocking* or *swap-allowed blocking*. In the alternative graph, a solution may be feasible when there are no positive length cycles. If a zero length cycle is feasible, this is the case of cycles of *swap-allowed-blocking* operations. In the case of cycles of *no-swap-blocking* operations, we can put a small positive weight on the corresponding alternative arcs, to model the fact that a job move strictly after the subsequent resource becomes available. By so doing, we can make infeasible a solution containing cycles of alternative arcs.

To be deadlock-free for the *no-swap-blocking* problems, it basically requires that at any moment, the set of jobs does not contain a so-called "*Hold-While-Wait*" cycle. The deadlock cycle may occur at a moment if a job J_1 holds machine M_1 while waiting for M_2 for progressing, a job J_2 holds M_2 while waiting for M_3 for progressing, and so on, to job J_n which holds M_n while waiting for M_1 for progressing. In case the "*Hold-While-Wait*" cycle is unavoidable, the deadlock-free status may be guaranteed only when the resources (e.g. machines) are available in multiple units. This situation typically happens in train scheduling problems.

5.5 Topological-Sequence Algorithm

5.5.1 Introduction of Algorithm

The topological-sequence algorithm was initially proposed by Liu and Ong (2002) to calculate the starting times and delivery times for solving the classical *permutation and general (non-permutation) flow-shop scheduling* (PFSS and GFSS) problems. Later, embedded in metaheuristics, this algorithm is a keystone in solving many types of multi-stage scheduling problems, namely, the *static* classical *job-shop, open-shop, mixed-shop and group-shop scheduling* (JSS, OSS, MSS and GSS) and *dynamic shop scheduling* (DSS) problems (Liu and Ong 2002; Liu and Ong; 2004; Liu *et al.* 2005a; Liu *et al.* 2005b).

In the literature, the topological-sequence algorithm was based on the disjunctive graph and designed for the scheduling environment with unlimited storage buffer. With exploring the characteristics of blocking conditions based on the alternative graph, a modified topological-sequence algorithm is developed for solving the *non-classical* shop scheduling problems without storage buffer.

5.5.2 Procedure of Algorithm

The procedure of the topological-sequence algorithm is described below.

Step 1: Based on the directed alternative graph model, compute the in-count value (i.e. the number of predecessors) of each node in the alternative graph; set the list of predecessors and successors of each node;

Step 2: Find a topological sequence of the total nodes in the alternative graph:

2.1 Select node 0 as the first node on the topological order list.

2.2 Decrease the in-count value for each of the immediate successor nodes of the selected node by one.

2.3 Select any of the unselected nodes having a zero in-count value and put this node as the next node on the topological order list.

2.4 Repeat Steps 3.2 and 3.3 until all nodes are selected.

Step 3: Initialisation, e.g. set the starting times and blocking times of all nodes as zero.

Step 4: From each node in the topological sequence, do:

Set the index of operation: i.

If operation O_i is processed on the last machine, the two predecessors of operation O_i are operations $O_{PJ[i]}$ and $O_{PM[i]}$. The starting time of operation O_i is calculated by Eq.(5.12):

$$e_i = \max(e_{PM[i]} + p_{PM[i]}, e_{PJ[i]} + p_{PJ[i]}) \tag{5.12}$$

Otherwise, the two predecessors of operation O_i are operations $O_{PJ[i]}$ and $O_{SJ[PM[i]]}$. In this case, the starting time of operation O_i is calculated by Eq.(5.13):

$$e_i = \max(e_{SJ[PM[i]]}, e_{PJ[i]} + p_{PJ[i]}) \tag{5.13}$$

Step 5: From each node in the topological sequence, do:

Set the index of operation: i.

If operation O_i is not processed on the last machine, the two successors of operation O_i are operations $O_{SJ[i]}$ and $O_{SM[PJ[i]]}$; thus the blocking time of O_i is obtained by Eq. (5.14):

$$b_i = e_{SJ[i]} - (e_{[i]} + p_{[i]}) \tag{5.14}$$

5.5.3 Illustration of Algorithm

The proposed algorithm is illustrated by a BFSS example with four jobs and three machines. The processing times of this test problem are given in Table 5-3.

Table 5-3: The processing times of a BFSS example

	M_1	M_2	M_3
J_1	$p_1 = 1$	$p_5 = 3$	$p_9 = 3$
J_2	$p_2 = 1$	$p_6 = 2$	$p_{10} = 2$
J_3	$p_3 = 4$	$p_7 = 1$	$p_{11} = 4$
J_4	$p_4 = 2$	$p_8 = 2$	$p_{12} = 2$

If the given permutation schedule of jobs is $\{J_2, J_3, J_1, J_4\}$, the directed alternative graph model for this BFSS problem is build up in Figure 5-13.

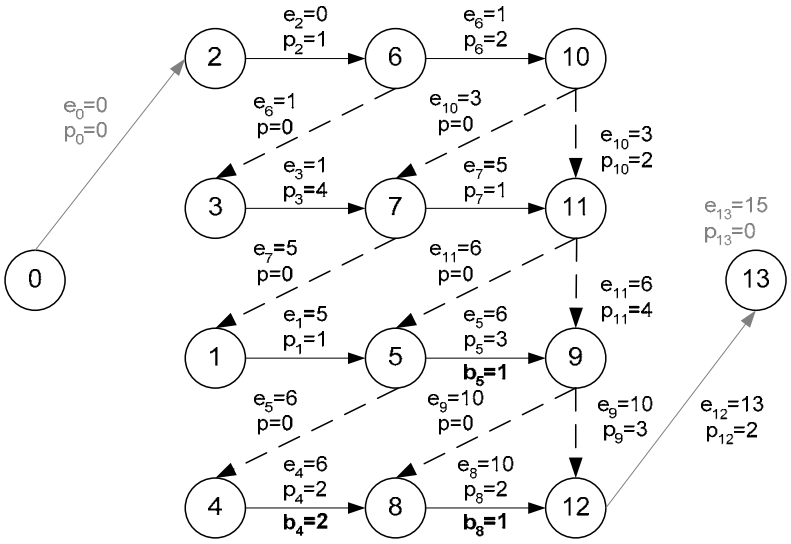

Figure 5-13: The directed alternative graph for a 4-job 3-machine BFSS example

In Figure 5-13, every node has at most two predecessors and at most two successors. In addition, the dummy start (finish) node connects the first (last)-processing operation in the graph. Moreover, if one operation O_i is not processed on the last machine, its two predecessors may consist of $O_{PJ[i]}$ and $O_{SJ[PM[i]]}$. For example, the predecessors of operation O_5 are: $O_{PJ[5]} = O_1$ and $O_{SJ[PM[5]]} = O_{SJ[7]} = O_{11}$. On the other hand, its two successors may comprise $O_{SJ[i]}$ and $O_{SM[PJ[i]]}$. For example, the successors of operation O_5 are: $O_{SJ[5]} = O_9$ and $O_{SM[PJ[5]]} = O_{SM[1]} = O_4$.

After applying the proposed topological-sequence algorithm to solve this graph model, the topological sequence of total nodes is obtained: {O_0, O_2, O_6, O_{10}, O_3, O_7, O_{11}, O_1, O_5, O_9, O_4, O_8, O_{12}, O_{13}}. Using equations (5.12) and (5.13), the staring time of each node can be determined in this topological sequence. For example, for node 5, the starting time is computed by Eq. (5.12):

$$\begin{aligned} e_5 &= \max(e_{SJ[PM[5]]}, e_{PJ[5]} + p_{PJ[5]}) \\ &= \max(e_{11}, e_1 + p_1) = \max(6, 5+1) = 6 \end{aligned}$$

For node 9 that is processed on the last machine, Eq. (5.13) is applied instead of Eq. (5.12) to calculate the starting time:

$$\begin{aligned} e_9 &= \max(e_{PM[i]} + p_{PM[7]}, e_{PJ[9]} + p_{PJ[9]}) \\ &= \max(e_{11} + p_{11}, e_5 + p_5) = \max(6+4, 6+3) = 10 \end{aligned}$$

Actually, the blocking time of a blocking operation O_i is equal to the gap between the completion time of O_i and the starting time of its same-job successor $O_{SJ[i]}$. After determining the starting times of all nodes, the blocking times for the operations that block a certain machine can be easily determined. For example, for node 8, its blocking time can be computed by Eq. (5.14):

$$\begin{aligned} b_8 &= e_{SJ[8]} - (e_{PJ[8]} + p_{PJ[8]}) \\ &= e_{12} - (e_8 + p_8) = 13 - (10+2) = 1 \end{aligned}$$

Our application of the proposed topological-sequence algorithm based on the alternative graph for solving the BFSS problem has been coded by Visual C++. The solution can be displayed and analysed in computer graphic interfaces. After running the proposed algorithm, one feasible solution of this test problem is displayed by the Gantt chart in Figure 5-14.

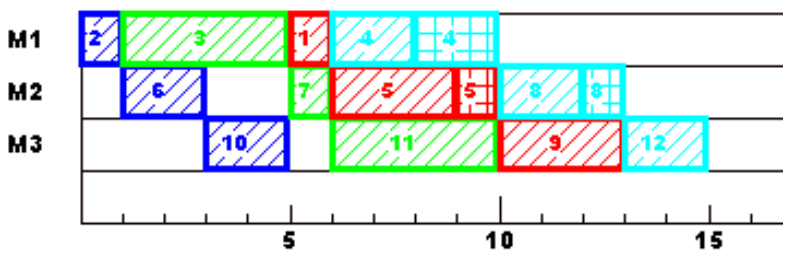

Figure 5-14: Gantt chart for a feasible solution of a 4-job 3-machine BFSS example, in which the blocking times are highlighted by cross brush

In Figure 5-14, the box brushed by the *45 degree upward hatch* is used to indicate the starting time and finishing time of each operation on a certain machine. In

addition, if one operation blocks a machine, the blocking time is revealed by the box brushed by the *cross hatch*. For example, the operation 5, having completed on M_2 at time point 9, has to remain on M_2 till the downstream machine M_3 becomes available for processing the operation 9 (its same-job successor) at time point 10.

To further validate the efficiency of the proposed topological-sequence algorithm, extensive computational experiments have been done to solve 120 (i.e. ta001-ta120) large-size flow-shop benchmark instances (Taillard, 1993). For example, one feasible solution of 20-job 20-machine ta021 instance with blocking is obtained and displayed by Gantt chart in Figure 5-15.

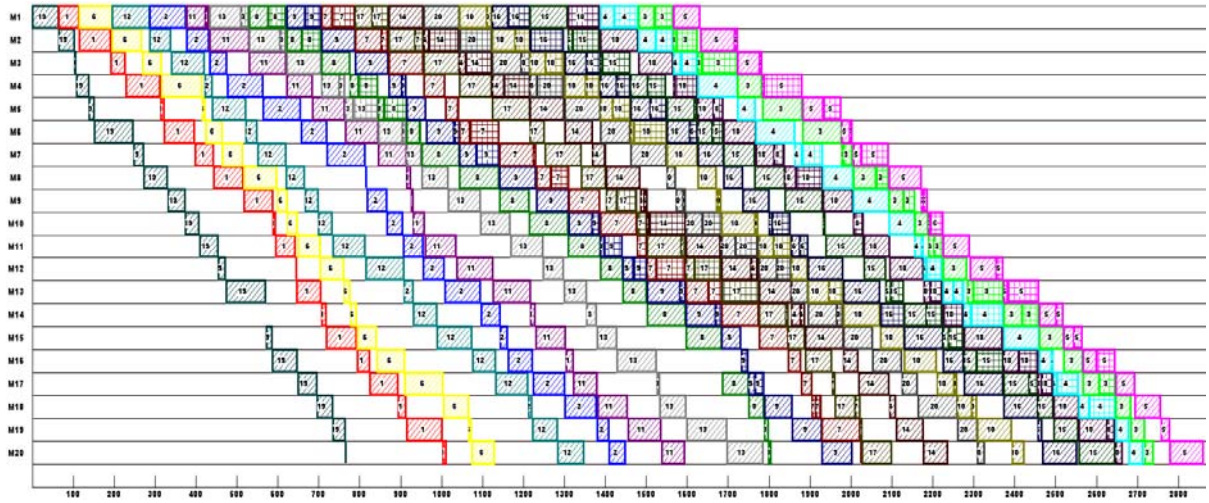

Figure 5-15: Gantt chart for the feasible solution of a 20-job 20-machine BFSS instance

5.5.4 Preeminence of Algorithm

As discussed above, there are two typical algorithms in the literature that can be applied to determine the makespan of a feasible BFSS schedule. To indicate the preeminence of the proposed *topological-sequence algorithm*, an attempt is made to compare it with these two approaches: *Directed Graph* and *Recursive Procedure*.

One way of obtaining the makespan of a given schedule is to find the longest path through a single machine or a node in a so-called *directed graph* (Pinedo 1995 and Ronconi 2004), as discussed in Section 5.3.2.

The length of a particular path connecting those nodes is a lower bound of makespan. The length of the longest path through a machine M_k is calculated by the following formula:

$$PL(k) = \sum_{q=1}^{k} p_{\pi_1,q} + \sum_{r=2}^{n} \max\{p_{\pi_r,k}, p_{\pi_{r-1},k+1}\} + \sum_{q=k+1}^{m} p_{\pi_n,q}$$

where $p_{\pi_j,m+1} = 0$ for every $j = 1,\ldots,n$. For example, using the above 4-job 3-machine BFSS instance, $PL(2)$ is the length of the longest path through machine M_2 and the calculation procedure is given below:

$$\begin{aligned} PL(2) &= (p_{21} + p_{22}) + (\max\{p_{3,2}, p_{2,3}\} + \max\{p_{1,2}, p_{3,3}\} + \max\{p_{4,2}, p_{1,3}\}) + p_{43} \\ &= (1+2) + (\max\{1,2\} + \max\{3,4\} + \max\{2,3\}) + 2 \\ &= 14 \end{aligned}$$

Furthermore, the path does not necessarily pass through all nodes related to a single machine. The length of the longest path passing through a node, which belongs to job π_j and is processed on machine M_k, can be computed by

$$PL(\pi_j, k) = \sum_{q=1}^{k} p_{\pi_1,q} + \sum_{r=2}^{j} \max\{p_{\pi_r,k}, p_{\pi_{r-1},k+1}\} + p_{\pi_j,k+1} + \sum_{r=j+1}^{n} \max\{p_{\pi_r,k+1}, p_{\pi_{r-1},k+2}\} + \sum_{q=k+2}^{m} p_{\pi_n,q}$$

where $p_{\pi_j,m+1} = p_{\pi_j,m+2} = 0$ for every $j = 1,\ldots,n$. For example, the length of the longest path through node $(1,1)$ is calculated as follows:

$$\begin{aligned} PL(1,1) &= p_{21} + (\max\{p_{3,1}, p_{2,2}\} + \max\{p_{1,1}, p_{3,2}\}) + p_{1,2} + \max\{p_{4,2}, p_{1,3}\} + (p_{4,3} + p_{4,4}) \\ &= 1 + (\max\{1,2\} + \max\{1,1\}) + 3 + \max\{2,3\} + (2+0) = 12 \end{aligned}$$

Therefore, we can obtain several lower bounds of makespan, formed by paths that pass through different machines or different nodes in directed graph. However, the length values of these paths are usually smaller than the makespan, which means that the paths obtained are not the critical (longest) path. For example, the lengths of the above two paths are respectively 14 and 12, which are smaller than the makespan value of 15, i.e. the length of the critical (longest) path.

In comparison, the proposed topological-sequence algorithm can efficiently calculate the starting times and the blocking times of all nodes. Then, the critical path and its length equal to the makespan can be correspondingly obtained.

The other approach is a *recursive-procedure* algorithm that recursively calculates the departure times of each job on each machine. From Appendix 2, the calculation procedure for a 4-job 3-machine BFSS example is summarised in Table 5-4.

Table 5-4: The calculation procedure of recursive-procedure algorithm

J_2	$D_{2,0}=0, D_{2,1}=1,\ D_{2,2}=3, D_{2,3}=5$
J_3	$D_{3,0}=D_{2,1}=1, D_{3,1}=\max(D_{3,0}+p_{3,1}, D_{2,2})=5,$ $D_{3,2}=\max(D_{3,1}+p_{3,2}, D_{2,3})=6, D_{3,3}=D_{3,2}+p_{3,3}=10$
J_1	$D_{1,0}=D_{3,1}=5, D_{1,1}=\max(D_{1,0}+p_{1,1}, D_{3,2})=6,$ $D_{1,2}=\max(D_{1,1}+p_{1,2}, D_{3,3})=10, D_{1,3}=D_{1,2}+p_{1,3}=13$
J_4	$D_{4,0}=D_{1,1}=6, D_{4,1}=\max(D_{4,0}+p_{4,1}, D_{1,2})=10,$ $D_{4,2}=\max(D_{4,1}+p_{4,2}, D_{1,3})=13, D_{4,3}=D_{4,2}+p_{4,3}=15$

Although the recursive-procedure method is very efficient, it is only applicable to the *permutation* BFSS schedule. In comparison with the recursive-procedure algorithm, the proposed topological-sequence algorithm is generic and adaptive, because it can be applied to solve either *the permutation BFSS*, *the general (non-permutation) BFSS* or *the BJSS (blocking job-shop problem)* problem, only if the given directed alternative graph is acyclic. The preeminence of the proposed algorithm is validated as below.

While using the above BFSS example shown in Figure 5-13, the *BFSS* schedule is changed to be a *BJSS* schedule by adjusting the machine sequence of job J_4 (i.e. 4→8→12 becomes 4→12→8). This obtained *BJSS* schedule can be modelled by a directed alternative graph model shown in Figure 5-16.

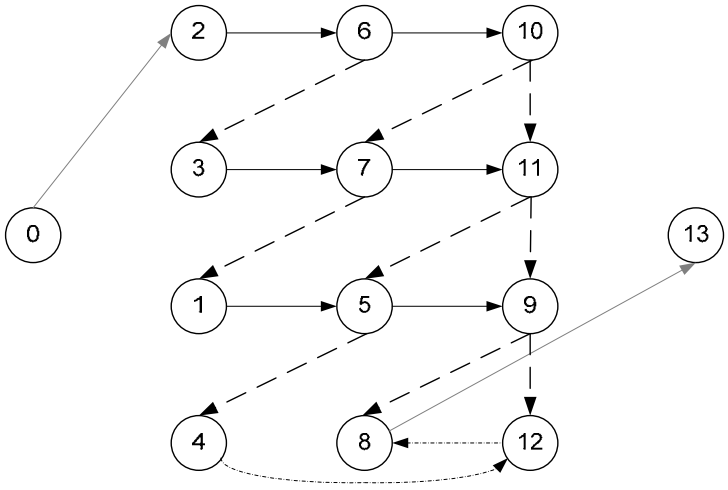

Figure 5-16: A directed alternative graph for a 4-job 3-machine *BJSS* schedule

After applying the proposed topological-sequence algorithm, the feasible solution for this *BJSS* schedule given in Figure 5-16 can be obtained and displayed by the Gantt chart in Figure 5-17. It is observed that the topological sequence is changed as: $\{O_0, O_2, O_6, O_{10}, O_3, O_7, O_{11}, O_1, O_5, O_9, O_4, O_{12}, O_8, O_{13}\}$.

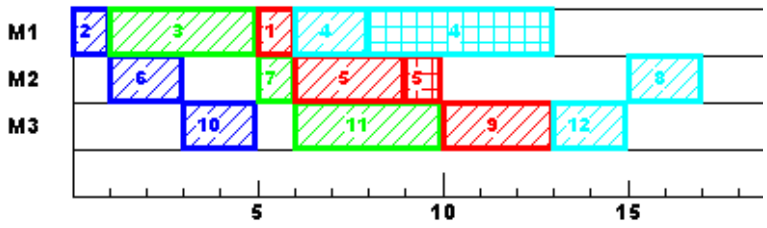

Figure 5-17: Gantt chart for the feasible solution of a *BJSS* schedule

To further prove that the proposed topological-sequence algorithm is generic and adaptive, the *BJSS* schedule shown in Figure 5-16 is altered again by changing the machine sequence of job J_3 (i.e. 3→7→11 becomes 3→11→7). Based on the directed alternative graph in Figure 5-18 for this new *BJSP* schedule, the solution with the new topological sequence $\{O_0, O_2, O_6, O_{10}, O_3, O_{11}, O_7, O_1, O_5, O_9, O_4, O_{12}, O_8, O_{13}\}$ is displayed by the Gantt chart in Figure 5-19.

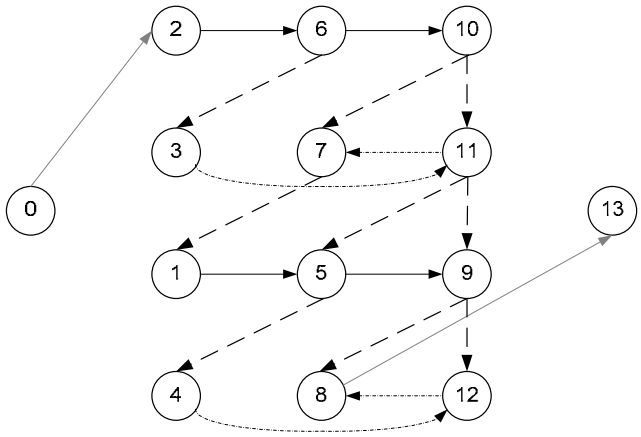

Figure 5-18: A directed alternative graph for a new *BJSS* schedule

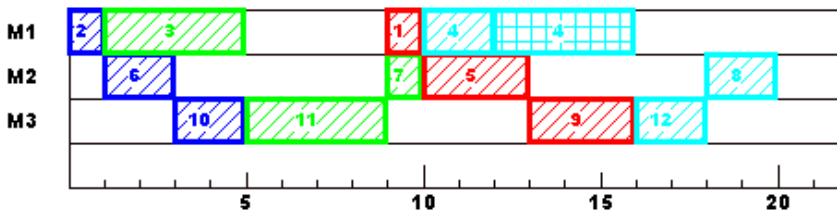

Figure 5-19: Gantt chart for the feasible solution of a new *BJSS* schedule

Furthermore, in comparison with these two known approaches, the proposed topological-sequence algorithm is able to quickly examine whether the given schedule is feasible or not. Extensive computational experiments indicate that the directed alternative graph is definitely acyclic (infeasible) if the topological sequence cannot be found; and that the program running can stop immediately in Step 2 of the proposed topological-sequence algorithm.

For more details about the proposed topological-sequence algorithm applied to non-classical shop scheduling problems, please refer to Liu and Kozan (2007a).

5.6 Combined-Buffer Flow-Shop Scheduling

5.6.1 Introduction of CBFSS

This section considers a new type of *non-classical* scheduling problem, the flow-shop scheduling (FSS) problem with combined buffer conditions including *No-Wait*, *No-Buffer* (*Blocking*), *Limited-Buffer* and *Infinite-Buffer*. For convenience, we call this class of scheduling problems *Combined-Buffer Flow-Shop Scheduling* (CBFSS) problem, which covers the classical FSS, the *blocking* FSS (BFSS), the *no-wait* FSS (NWFSS) and the *limited-buffer* FSS (LBFSS) problems.

As far as we know, no research work is devoted to the CBFSS problem which considers four buffer conditions (i.e. *No-Wait*, *No-Buffer, Limited-Buffer* and *Infinite-Buffer*) simultaneously. In this research, an innovative constructive algorithm called the *LK* algorithm is developed for solving this problem by exploiting the structural properties of the combined inter-machine buffer conditions (Liu and Kozan, 2007b).

5.6.2 Definition of CBFSS

In a flow shop with m machines and n jobs, let $b_{j,j+1}$ represent the buffer condition between two successive machines M_j and M_{j+1} ($\forall j = 1,2,\ldots,m-1$). Usually, $b_{j,j+1}$ is interpreted as the fixed maximum amount of buffer storage or the maximum number of jobs that can be stored in the buffer. However, the following discourse is confined to problems with buffers that are assigned to pairs of two consecutive machines since the buffer $b_{j,j+1}$ is only able to store jobs that finish on M_j and wait for their processing on M_{j+1}. In addition, it is supposed that the capacity of buffers is infinite in front of the first machine M_1 and behind the last machine M_m.

Thus, to respectively represent the buffer condition between one pair of two successive machines, the possible values for $b_{j,j+1}$ are:

$$b_{j,j+1} \in B \equiv \{-1,0,1,2,\ldots,\infty\},\ \forall j = 1,\ldots,m-1$$

- $b_{j,j+1} = 0$ if there is no buffer storage between M_j and M_{j+1}. This buffer condition is called "*No-Buffer*". If $b_{j,j+1} = 0$ $(\forall j = 1,\ldots,m-1)$, there is no buffer storage between any two successive machines. Thus, the availability of machine M_j is dependent on its downstream machine M_{j+1}. If a job J_1 finishes on M_j and the processing of another job J_2 on M_{j+1} has not been finished yet, J_1 blocks M_j until M_{j+1} becomes available. In this case, this problem is treated as the *blocking flow-shop scheduling* (BFSS) problem.

- $b_{j,j+1} = -1$ when no in-process interruption is allowed between starting a job on M_j and finishing it on M_{j+1}. This buffer condition is called "*No-Wait*". If $b_{j,j+1} = -1$ $(\forall j = 1,\ldots,m-1)$, all jobs have to be processed without any interruption from their start on the first machine until their completion on the last machine. Obviously, the capacity of buffer storage between any two consecutive machines need not be considered. Moreover, the above mentioned "*blocking*" conditions are absolutely not allowed in this case. For this reason, a more severe constraint to the problem than the blocking constraint appears here.

In the literature, this special class of problem is interpreted as the *no-wait (or continuous) flow-shop scheduling* (NWFSS) problem.

- $b_{j,j+1} = \infty$ if there is no upper bound for the capacity of buffer storage between machine M_j and M_{j+1}. This buffer condition is called "*Infinite-Buffer*". If $b_{j,j+1} = \infty$ $(\forall j = 1,\ldots,m-1)$, the problem is known as the *classical flow-shop scheduling* (FSS) problem.

- $b_{j,j+1} \in \{1,2,\ldots,\Theta\}$ if the capacity of buffers between machine M_j and M_{j+1} is limited. This buffer condition is called "*Limited-Buffer*". Here, Θ is defined as the realistic maximum capacity of buffers needed to store the jobs and usually it is redundant if Θ exceeds n. If $b_{j,j+1} \in \{1,2,\ldots,\Theta\}$ and when this limited-capacity buffer between machine M_j and M_{j+1} is exhausted at a certain time, what would happen if a job J_1 finishes on M_j and the processing of another job J_2 on M_{j+1} has not finished yet? It is suitable to suppose that J_1 blocks M_j until it would be forwarded into one buffer with the earliest available time between M_j and M_{j+1}. In this case, the availability of machine M_j is dependent not only on the availability of the downstream machine M_{j+1} but also on the availability of buffer between M_j and M_{j+1}. If $b_{j,j+1} \in \{1,2,\ldots,\Theta\}$ $(\forall j = 1,\ldots,m-1)$, this special case is regarded as the *limited-buffer flow-shop scheduling* (LBFSS) problem.

If $b_{j,j+1} \in \{-1,0,1,2,\ldots,\infty\}$ $(\forall j = 1,\ldots,m-1)$, the buffer condition between two successive machines can be any one of four buffer conditions analysed above, varied with the value of $b_{j,j+1}$. Thus, the *combined-buffer flow-shop scheduling* (CBFSS) problem is defined here. For convenience, this class of scheduling problem can be represented by the 4-field descriptor, $\alpha \mid \beta \mid \theta \mid \gamma$ discussed in Chapter 3. For example, a *m*-machine *n*-job CBFSS problem is denoted as:

$$Fn \mid m \mid b_{j,j+1} \in \{-1,0,1,2,\ldots,\infty\}, \forall j = 1,\ldots,m-1 \mid C_{\max},$$

where $b_{j,j+1} \in \{-1,0,1,2,\ldots,\infty\}$ defines the buffer condition between two successive machines M_j and M_{j+1} in terms of various values; F represent a flow shop; $C_{\max}$ is the objective function to minimise the makespan.

5.6.3 Solution Techniques for CBFSS

5.6.3.1 Notations of LK Algorithm

The proposed constructive heuristic, called the *LK* Algorithm, is designed by exploiting the structural properties of the above four buffer conditions (i.e. "*No-Wait*", "*No-Buffer*", "*Limited-Buffer*", and "*Infinite-Buffer*") to obtain the feasible solution of one CBFSS schedule. The *LK* algorithm mainly consists of two procedures, namely, *time-determination procedure* and *tune-up procedure*. In addition, in terms of distinct buffer conditions, eight sets of formulae are respectively developed and used for time-determination procedure and tune-up procedure in the *LK* algorithm.

Some notations that will be used by the *LK* algorithm are defined as follows:

O_k operation k processed on machine M_j

$O_{PJ[k]}$ the same-job predecessor of O_k ; obviously, $O_{PJ[k]}$ is processed on M_{j-1} in a flow shop.

$O_{SJ[k]}$ the same-job successor of O_k ; obviously, $O_{SJ[k]}$ is processed on M_{j+1} in a flow shop.

$O_{PM[k]}$ the same-machine predecessor of O_k

$O_{SM[k]}$ the same-machine successor of O_k

$O_{SJ[PM[k]]}$ the same-job successor of operation $O_{PM[k]}$

$b_{j,j+1}$ the buffer condition between machines M_j and M_{j+1}

$A^i_{b_{j,j+1}}$ the earliest available time of the i^{th} $(i = 1,2,\ldots,b_{j,j+1} \mid b_{j,j+1} \in \{1,\ldots,\Theta\})$ buffer between M_j and M_{j+1}

$A^*_{b_{j,j+1}}$ the earliest available time of the buffers between M_j and M_{j+1}; obviously, $A^*_{b_{j,j+1}} = \min_{i\in\{1,2,..,b_{j,j+1}\}} A^i_{b_{j,j+1}} = A^x_{b_{j,j+1}}$, assuming that the x^{th} buffer is selected as it leads to the earliest available time at current stage.

A_k the earliest available time of the selected buffer that stores operation O_k; obviously, $A_k = A^*_{b_{j,j+1}}$.

e_k the starting time of O_k

p_k the processing time of O_k

c_k the completion time of O_k ($c_k = e_k + p_k$)

B_k the blocking time of O_k

D_k the departure time of O_k ($D_k = c_k + B_k$)

S_k the storing time of O_k

e'_k the updated starting time of O_k after tune-up

5.6.3.2 Time-Determination Procedure

In time-determination procedure, with satisfying the given buffer condition between M_j and M_{j+1}, represented by the value of $b_{j,j+1}$, the starting time, completion time, blocking time, departure time and storing time of operation O_k processed on M_j can be sequentially determined by employing the following corresponding set of formulae.

- If the buffer condition between M_j and M_{j+1} is *No-Wait* (i.e. $b_{j,j+1} = -1$), the formulae Set (1) is applied:

$$
\begin{aligned}
e_k &= \begin{cases} \max(D_{SJ[PM[k]]} - p_k, D_{PJ[k]}) & \text{if } D_{SJ[PM[k]]} - p_k \geq D_{PM[k]} \\ \max(D_{PM[k]}, D_{PJ[k]}) & \text{if } D_{SJ[PM[k]]} - p_k < D_{PM[k]} \end{cases} \\
c_k &= e_k + p_k \\
B_k &= 0 \\
D_k &= c_k + B_k \\
S_k &= 0
\end{aligned}
\qquad \text{Set (1)}
$$

- If the buffer condition between M_j and M_{j+1} is *No-Buffer* (i.e. $b_{j,j+1} = 0$), the formulae Set (2) is applied:

$$
\begin{aligned}
& e_k = \max(D_{PM[k]}, D_{PJ[k]} + S_{PJ[k]}) \\
& c_k = e_k + p_k \\
& B_k = \begin{cases} D_{SJ[PM[k]]} - c_k & \text{if } D_{SJ[PM[k]]} > c_k \\ 0 & \text{if } D_{SJ[PM[k]]} \le c_k \end{cases} \\
& D_k = c_k + B_k \\
& S_k = 0
\end{aligned}
\qquad \text{Set (2)}
$$

- If the buffer condition between M_j and M_{j+1} is *Limited-Buffer* (i.e. $b_{j,j+1} \in \{1, 2, \ldots, \Theta\}$), the formulae Set (3) is applied:

$$
\begin{aligned}
& e_k = \max(D_{PM[k]}, D_{PJ[k]} + S_{PJ[k]}) \\
& c_k = e_k + p_k \\
& A_k = A^*_{b_{j,j+1}} = A^k_{b_{j,j+1}} = \min_{i \in \{1,2,..,b_{j,j+1}\}} A^i_{b_{j,j+1}} \\
& B_k = \begin{cases} 0 & \text{if } D_{SJ[PM[k]]} \le c_k \\ 0 & \text{if } D_{SJ[PM[k]]} > c_k \text{ and } A_k \le c_k \\ A_k - c_k & \text{if } D_{SJ[PM[k]]} > c_k \text{ and } A_k > c_k \end{cases} \\
& D_k = c_k + B_k \\
& S_k = \begin{cases} 0 & \text{if } D_{SJ[PM[k]]} \le D_k \\ D_{SJ[PM[k]]} - D_k & \text{if } D_{SJ[PM[k]]} > D_k \end{cases} \\
& A^k_{b_{j,j+1}} = \max(A^*_{b_{j,j+1}}, D_k + S_k)
\end{aligned}
\qquad \text{Set (3)}
$$

- If the buffer condition between M_j and M_{j+1} is *Infinite-Buffer* (i.e. $b_{j,j+1} = \infty$), the formulae Set (4) is applied:

$$
\begin{aligned}
& e_k = \max(D_{PM[k]}, D_{PJ[k]} + S_{PJ[k]}) \\
& c_k = e_k + p_k \\
& B_k = 0 \\
& D_k = c_k + B_k \\
& S_k = 0
\end{aligned}
\qquad \text{Set (4)}
$$

5.6.3.3 Tune-Up Procedure

To satisfy the no-wait buffer conditions between all pairs of two successive operations, the start of a job on a certain machine has to be postponed until the corresponding operation's completion coincides with the start of the next operation on the subsequent machine.

Assuming that operation $O_{PJ[k]}$ is processed on upstream machine M_{j-1}, one situation in the tune-up procedure is illustrated in Figure 5-20. By analysis, the starting time of $O_{PJ[k]}$ may need to be updated in order to satisfy the no-wait conditions. Compared to the time-determination procedure, the tune-up procedure is in a backward order.

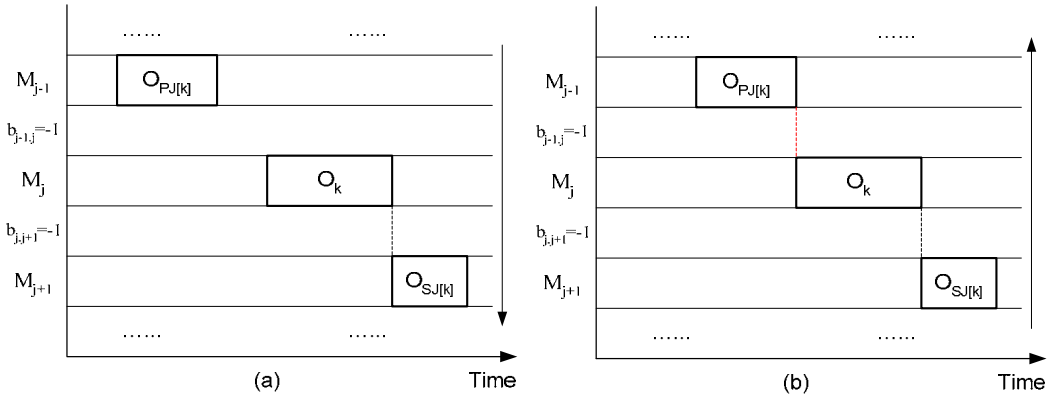

Figure 5-20: Comparison of (a) time-determination procedure and (b) tune-up procedure

In the same fashion, according to the given buffer condition between M_{j-1} and M_j, represented by the value of $b_{j-1,j}$, the starting time, completion time, blocking time, departure time and storing time of operation $O_{PJ[k]}$ processed on M_{j-1} are needed to be tuned up if $e_k \neq e'_k$ or $b_{j-1,j} = -1$, by implementing the following set of formulae correspondingly.

- In tune-up procedure, if the buffer condition between M_{j-1} and M_j is *No-Wait* (i.e. $b_{j-1,j} = -1$), the formulae Set (5) is applied:

$$
\begin{aligned}
e'_{PJ[k]} &= e'_k - p_{PJ[k]} \\
c'_{PJ[k]} &= e'_k \\
B'_{PJ[k]} &= 0 \\
D'_{PJ[k]} &= c'_{PJ[k]} + B'_{PJ[k]} \\
S'_{PJ[k]} &= 0
\end{aligned}
\qquad \text{Set (5)}
$$

- In tune-up procedure, if the buffer condition between M_{j-1} and M_j is *No-Buffer* (i.e. $b_{j-1,j} = 0$), the formulae Set (6) is applied:

$$
\begin{aligned}
e'_{PJ[k]} &= e_{PJ[k]} \\
c'_{PJ[k]} &= c_{PJ[k]} \\
B'_{PJ[k]} &= \max(0, e'_k - e'_{PJ[k]}) \\
D'_{PJ[k]} &= c'_{PJ[k]} + B'_{PJ[k]} \\
S'_{PJ[k]} &= 0
\end{aligned}
\qquad \text{Set (6)}
$$

- In tune-up procedure, if the buffer condition between M_{j-1} and M_j is *Limited-Buffer* (i.e. $b_{j-1,j} \in \{1, 2, \ldots, \Theta\}$), the formulae Set (7) is applied:

$$
\begin{aligned}
e'_{PJ[k]} &= e_{PJ[k]} \\
c'_{PJ[k]} &= c_{PJ[k]} \\
A'_{PJ[k]} &= A^x_{b_{j-1,j}} - S_{PJ[k]} \\
B'_{PJ[k]} &= \begin{cases} A'_{PJ[k]} - c'_{PJ[k]} & \text{if } e'_k > c'_{PJ[k]} \text{ and } A'_{PJ[k]} > c'_{PJ[k]} \\ 0 & \text{otherwise} \end{cases} \\
D'_{PJ[k]} &= c'_{PJ[k]} + B'_{PJ[k]} \\
S'_{PJ[k]} &= \begin{cases} e'_k - D'_{PJ[k]} & \text{if } e'_k \geq D'_{PJ[k]} \\ 0 & \text{otherwise} \end{cases} \\
A^x_{b_{j-1,j}} &= A'_{PJ[k]} + S'_{PJ[k]}
\end{aligned}
\qquad \text{Set (7)}
$$

- In tune-up procedure, if the buffer condition between M_{j-1} and M_j is *Infinite-Buffer* (i.e. $b_{j-1,j} = \infty$), the formulae Set (8) is applied:

$$
\begin{aligned}
e'_{PJ[k]} &= e_{PJ[k]} \\
c'_{PJ[k]} &= c_{PJ[k]} \\
B'_{PJ[k]} &= 0 \\
D'_{PJ[k]} &= c'_{PJ[k]} + B'_{PJ[k]} \\
S'_{PJ[k]} &= e'_k - D'_{PJ[k]}
\end{aligned}
\qquad \text{Set (8)}
$$

5.6.3.4 Description of LK Algorithm

Based on the above eight sets of formulae, the *LK* algorithm is proposed for solving the CBFSS problem and briefly described below:

For each job in the given permutation sequence, do:

From the first machine to the last second machine, do:

Set the machine index: j; and its downstream machine: $j+1$.

Set the index of operation processed on machine j: k.

Switch according to the value of $b_{j,j+1}$ that represents the buffer condition between machine j and machine $j+1$:

If $b_{j,j+1}=-1$, apply Set (1) for the *No-Wait* case.

If $b_{j,j+1}=0$, apply Set (2) for the *No-Buffer* case.

If $b_{j,j+1}\in\{1,2,\ldots,\Theta\}$, apply Set (3) for the *Limited-Buffer* case.

If $b_{j,j+1}=\infty$, apply Set (4) for the *Infinite-Buffer* case.

Because the buffer condition behind the last machine is regarded as *Infinite-Buffer*, apply Set (4) for the operation processed on the last machine.

After determining all operations of each job, apply the tune-up procedure in a backward order from the last machine to the first machine. In the tune-up procedure, similarly switch according to the value of $b_{j,j+1}$:

If $b_{j,j+1}=-1$, apply Set (5) for the *No-Wait* case.

If $b_{j,j+1}=0$, apply Set (6) for the *No-Buffer* case.

If $b_{j,j+1}\in\{1,2,\ldots,\Theta\}$, apply Set (7) for the *Limited-Buffer* case.

If $b_{j,j+1}=\infty$, apply Set (8) for the *Infinite-Buffer* case.

5.6.3.5 Implementation of LK Algorithm

Our application of the proposed *LK* algorithm for the CBFSS problem has been coded by Visual C++. All typical application-specific modules are treated as the following objects: data, formulae, algorithms, output, interfaces and other

behaviours. Thus, it provides a convenient way to quickly get the new solutions of different problems by changing the attributes in data input, such as $b_{j,j+1}$ values that represent the various buffer conditions. Moreover, the solution can be displayed and analysed in a graphic interface.

The computational experiments of the proposed *LK* algorithm are given in detail in Appendix 3. It is validated that the proposed algorithm without modification is able to construct the feasible solution for any type of the following non-classical scheduling problems:

- the classical *permutation flow-shop scheduling* (PFSS);
- the *blocking flow-shop scheduling* (BFSS);
- *no-wait flow-shop scheduling* (NWFSS);
- *limited-buffer flow-shop scheduling* (LBFSS); and
- *combined-buffer flow-shop scheduling* (CBFSS).

The results demonstrate that the no-wait condition is the most restrictive constraint among these four different buffer conditions, and

5.6.3.6 LK-BIH Algorithm

The proposed *LK* algorithm is a *constructive algorithm* that aims to construct a feasible solution from scratch by a growth process till a completely feasible solution has been determined. Given an initial feasible (partial or complete) solution, a *local-search heuristic* like the *best-insertion-heuristic* (BIH) algorithm yields a set of neighbourhood solutions which converge to a preferable solution.

In order to build a better CBFSS solution, it is possible to define a two-stage ***hybrid heuristic*** by combining the ***constructive algorithm*** (e.g. the *LK* algorithm) and the ***local-search heuristic*** (e.g. the *BIH* algorithm) together. In this procedure, a *constructive heuristic* is applied to obtain the feasible (partial) alternative solution, which is improved by executing a *local-search heuristic* at each step. Thus, a two-stage hybrid heuristic algorithm called the *LK-BIH algorithm* is developed.

The procedure of the *LK-BIH* algorithm for solving the CBFSS problem is described as follows.

Step 1: Build up a partial solution M with only one initial job that has the longest processing time.

Step 2: For $k \leftarrow 2$ to $k \leftarrow n$, do

2.1: At step k, consider all possible combinations of $n-k+1$ jobs with k insertion positions to obtain a set of alternative sequences of k jobs.

2.2: Apply the *LK* algorithm to these alternatives and obtain the feasible CBFSS solutions.

2.3: Select M'^{*} that is the best CBFSS solution with the minimum makespan.

2.4: Update the partial solution, i.e. $M = M'^{*}$ and $k = k+1$. If M is complete, stop; otherwise, go to Step 2 for the next iteration.

In a sense, the *LK-BIH* algorithm considers all possible insertions of all not-yet-included jobs while successively building a complete CBFSS schedule. That is, starting with an initial job that has the longest processing time, at each step $k = 2,...,n$, all possible combinations of $n-k+1$ jobs with k insertion positions are investigated by the *LK* algorithm.

5.7 Summary

In this chapter, the non-classical scheduling problems including the BFSS, NWFSS, LBFSS and BJSS are reviewed. Some typical algorithms that exploit the characteristics of non-classical scheduling problems are described. The definition and analysis of the alternative graph are given in detail. The alternative graph as an extension of the classical disjunctive graph model is especially designed for modelling the *non-classical* scheduling problems.

By exploring the blocking characteristics based on an alternative graph, a topological-sequence algorithm is developed for solving the non-classical scheduling problems. To indicate the preeminence of the proposed algorithm, it is compared with two typical algorithms (i.e. *Recursive Procedure* and *Directed*

Graph). The comparison reveals that the proposed algorithm has the following merits:

- The proposed algorithm can efficiently calculate the starting times and blocking times in the topological order; and then determine the makespan and the critical path. In comparison, the length values of the longest paths (through one machine or a node) determined by *Directed Graph* are usually smaller than the makespan, which means that these paths are not the critical path.
- The proposed algorithm is generic and adaptive, because without any modification it can be applied to solve the BPFSS, BGFSS or BJSS problem, only if the corresponding directed alternative graph is acyclic. In comparison, the *Recursive Procedure* approach can only deal with the BPFSS problem.
- The proposed algorithm is able to quickly examine whether the given schedule is feasible or infeasible, by checking the availability of the topological sequence in the alternative graph model.

Moreover, we define a new type of non-classical scheduling problem, called *combined-buffer flow-shop scheduling* (CBFSS) which covers its four extreme cases: the *classical* FSS (FSS) with infinite buffer, the *blocking* FSS (BFSS) with no buffer, the *no-wait* FSS (NWFSS) and the limited-buffer FSS (LBFSS). With exploring the structural properties of CBFSS, we propose a new and generic constructive algorithm called the *LK* algorithm, which is mainly comprised of time-determination procedure and tune-up procedure with eight sets of formulae. The implementation of the *LK* algorithm indicates that the proposed algorithm is able to solve the FSS, BFSS, NWFSS, LBFSS and CBFSS problems without modification. It is validated that the proposed *LK* algorithm is very promising as a useful tool for modelling and solving many real-world scheduling problems with combined buffer constraints. For example, in real-world train scheduling, blocking conditions should be considered due to the lack of buffer storage. In addition, no-wait conditions may arise when considering the passenger trains that should traverse continuously without any interruption. Thus, the CBFSS problem can be extended to the *no-wait blocking parallel-machine flow-shop* (or *job-shop*) *scheduling*

(NWBPMFSS or NWBPMJSS) problem for modelling complicated overtaking and crossing situations in train scheduling, which will be studied in the next chapter.

The proposed *LK* algorithm is a constructive algorithm, which aims to build a feasible solution. To construct a better CBFSS schedule in an economical and efficient way, a two-stage hybrid algorithm called *LK-BIH* algorithm is developed by combining the *LK* algorithm with the *best-insertion-heuristic* (BIH) algorithm. The solution quality of CBFSS can be further improved by using the metaheuristics like tabu search (TS) described in the next Chapter.

Chapter 6 Train Scheduling

CHAPTER OUTLINE

PUBLICATIONS ARISING FROM CHAPTER 6:

Liu, S. Q and Kozan E. (2007c). A Blocking Parallel-Machine Job-Shop-Scheduling Model for the Train Scheduling Problem. *The 8th Asia-Pacific Industrial Engineering and Management Systems Conference*, Kaohsiung, Taiwan, 10.1-10.10.

Liu, S. Q., & Kozan, E. (2008a). Scheduling trains as a blocking parallel-machine job shop scheduling problem. *Computers and Operations Research* (In Press).

Liu, S. Q., & Kozan, E. (2008b). A hybrid shifting bottleneck procedure algorithm combined with metaheuristics for the parallel-machine job-shop scheduling problem. *Journal of Scheduling* (Submitted).

Liu, S. Q., & Kozan, E. (2008c). Scheduling trains with priorities: a no-wait blocking parallel-machine job-shop scheduling model. *Transportation Science* (Submitted).

6.1 Introduction

All over the world railway management usually deal with train scheduling by maximising the railway capacity or minimising the total delays by using empirical or analytical methods, tested by themselves over long periods of time. These methods have to resort to taking advantage of priority rules or non-generic characteristics of the specialised cases. It is also very hard to indentify their borders between generic and special, feasible and infeasible, hard and easy, theoretical and practical.

This study aims to achieve a significant efficiency improvement in a generalised rail network on the basis of the development of standard modelling approaches and generic solution techniques, by utilising a "*toolbox*" of standard well-solved *classical* and *non-classical* scheduling problems that have been discussed in Chapters 4 and 5.

The rail network to be studied consists of a set of single-track sections and a set of multiple-track sections referred to as *Crossing Loops* (*Sidings*), as depicted in Figure 6-1.

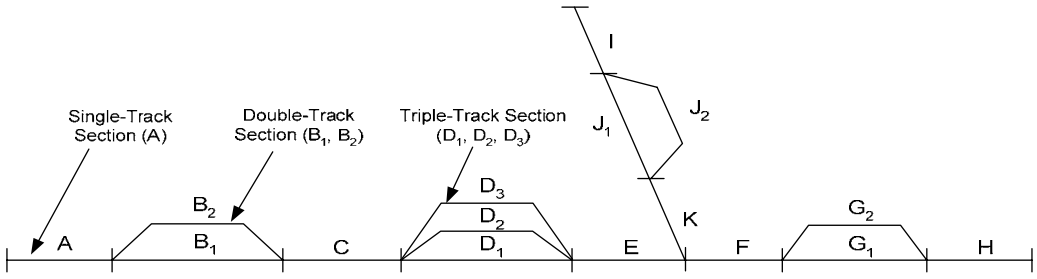

Figure 6-1: A railway network

In practice, the railway network should be such that only one train can occupy a single-track section at a time, whereas more than one train can be at a crossing loop (i.e. multiple-track section) at a time, provided that its capacity limit is regarded. Crossing loops are places where trains can stop or slow down in order to let another train overtake or cross it, or where trains can stop to load or unload cargoes, alight passengers and manoeuvre crew. Usually, a traversing track section (e.g. Section A or B_1 in Figure 6-1) is necessarily delimited by at least two signals: one at the

beginning and the other at the end of section, which will control when a train either can or cannot traverse on a section. This control is to avoid two trains running on the same traversing track simultaneously. Due to the lack of buffer or storage space, the real-world case should consider blocking or hold-while-wait constraints, which means that a track section cannot release and must hold the train until next section on the routing becomes available. As a consequence, the train scheduling problem should seriously consider the blocking conditions in process.

According to the above analysis, due to the limited-capacity buffers (i.e. blocking conditions) and parallel resources (multiple-track sections), the train scheduling problem can be basically modelled as a *Blocking Parallel-Machine Job-Shop Scheduling* (BPMJSS) problem (Liu and Kozan, 2008a). This is achieved by considering the train trips as jobs, which will be scheduled on single-track sections that are regarded as single machines, and on multiple-track sections that are referred to as parallel machines.

Moreover, no-wait conditions may arise when considering prioritised trains like express passenger trains that should traverse continuously without any interruption and any unplanned dwelling. In this case, for the sake of the high waiting cost for passengers in travelling, passenger trains must enter the next section immediately after traversing on the previous section is completed. Freight trains may be thought of as a relaxation of passenger trains as in comparison they may be allowed to remain in a section until the next section on the routing becomes available. In a sense, freight trains can be treated as *blocking* jobs while passenger trains are defined as *no-wait* jobs. The train scheduling problem, in which freight trains and passenger trains are considered simultaneously, can be regarded as a *No-Wait Blocking Parallel-Machine Job-Shop Scheduling* (NWBPMJSS) problem that is more generalised but more complicated than the BPMJSS problem.

Based on literature review and our analysis, it is very hard to directly find a feasible complete schedule for the BPMJSS or NWBPMJSS problem. Firstly, a *parallel-machine job-shop-scheduling* (PMJSS) problem without considering blocking constraints is solved by an improved *Shifting Bottleneck Procedure* (SBP) algorithm. Inspired by the proposed SBP algorithm, a *Feasibility Satisfaction*

Procedure (FSP) algorithm is developed to construct the feasible solution of the BPMJSS problem, by an alternative graph model that is an extension of the classical disjunctive graph models. Furthermore, a more generalised algorithm called the *SE* algorithm is proposed to solve the NWBPMJSS problem.

6.2 Parallel-Machine Job-Shop Scheduling

In this section, the train scheduling problem is temporarily modelled as a *parallel-machine job-shop scheduling* (PMJSS) problem without considering the *blocking* and *no-wait* constraints. In the model, trains, single-track sections and multiple-track sections respectively are synonymous with jobs, single machines and parallel machines; and an operation is regarded as the movement/traversal of a train across a section. The relationships are portrayed in more detail as follows.

- Jobs ↔ Trains
- Single Machines ↔ Single-Track Sections
- Parallel Machines ↔ Multiple-Track Sections
- Operations ↔ Train operations (The action of a train passing through a section is defined as a train operation.)
- Operational processing time ↔ Sectional running time

Thus for the rail network shown in Figure 6-1, the number of section types (or machine types) is 11, among which four sections (i.e. B, D, J, G) are multiple-track sections (parallel machines) and the other seven sections (i.e. A, C, E, F, I, K, H) are single-track sections (single machines). For example, we regard the double-track section (e.g. Section B) as one same-type machine with two units (e.g. B_1, B_2).

6.2.1 Mathematical Formulation for PMJSS

Notations

n number of jobs (trains).

m number of machines (sections).

J_i job i ($i = 1, 2, \ldots n$).

M_k machine k ($k = 1, 2, \ldots, m$).

h_k number of units of machine k; default is single machine $h_k = 1$.

u_l the l^{th} unit of machine k ($l = 1, \ldots, h_k$).

o index of sequence position of operation in one job ($o = 1, 2, \ldots, m$).

s_{ilk} starting time of job i on the l^{th} unit of machine k.

p_{ilk} processing time of job i on the l^{th} unit of machine k.

r_{iolk} = 1, if the o^{th} operation of job i requires the l^{th} unit of machine k;
= 0, otherwise.

x_{ilk} = 1, if job i is assigned to the l^{th} unit of machine k;
= 0, otherwise.

y_{ijlk} = 1, if both jobs i and j are assigned to the l^{th} unit of machine k and job i precedes job j (not necessarily immediately);
= 0, otherwise.

w_{ijolk} = 1, if the o^{th} operation of job i requires the l^{th} unit of machine k; and job j is scheduled on this same unit as its successor;
= 0, otherwise.

$C_{\max}$ maximum completion time or makespan.

L a very large positive number.

The mathematical programming formulation for PMJSS is proposed as follows:

PMJSS Model

$$Minimise \quad C_{\max} \tag{6.1}$$

The objective function is to minimise the makespan.

Subject to:

$$\sum_{l=1}^{h_k}\sum_{k=1}^{m} r_{iolk}(s_{ilk} + p_{ilk}) \le \sum_{l=1}^{h_k}\sum_{k=1}^{m} r_{i,o+1,l,k} s_{ilk}, o = 1, 2, \ldots, m-1, \forall i. \tag{6.2}$$

Equation (6.2) restricts the starting time of $(o+1)^{th}$ operation of job i to be no earlier than its finish time of the o^{th} operation of job i.

$$s_{ilk} \ge s_{jlk} + p_{jlk} + L(y_{ijlk} - 1) \ \forall i, j, l, k. \tag{6.3}$$

Equation (6.3) restricts that both jobs i and j are processed on the l^{th} unit of machine k and job i precedes job j (not necessarily immediately).

$$s_{jlk} \geq s_{ilk} + p_{ilk} + L(y_{jilk} - 1) \ \forall i, j, l, k. \tag{6.4}$$

Equation (6.4) restricts that that both jobs i and j are processed on the l^{th} unit of machine k and job j precedes job i (not necessarily immediately).

$$y_{ijlk} + y_{jilk} \leq 1 \ \forall i, j, l, k. \tag{6.5}$$

Equation (6.5) restricts that conditions that job j precedes job i or job i precedes job j at the l^{th} unit of machine k are exclusive.

$$\sum_{l=1}^{h_k} \sum_{k=1}^{m} x_{ilk} = 1 \text{ and } x_{ilk} + x_{jlk} - 1 \leq y_{ijlk} + y_{jilk} \ \forall i, j, l, k. \tag{6.6}$$

Equation (6.6) restricts that each unit can process at most one job at a time.

$$\sum_{l=1}^{h_k} \sum_{k=1}^{m} r_{imlk} (s_{ilk} + p_{ilk}) \leq C_{\max} \quad \forall i. \tag{6.7}$$

Equation (6.7) restricts that the completion time of the m^{th} (i.e. last) operation of each job is no earlier than makespan.

$$s_{ilk}, p_{ilk} \geq 0 \ \forall i, l, k. \tag{6.8}$$

Equation (6.8) satisfies non-negativity condition.

6.2.2 Disjunctive Graph Model for PMJSS

For better illustrating the disjunctive graph model, assume that five outbound trains and four inbound trains are running in the above railway network. The corresponding traversing routes for each outbound and inbound train are given in Table 6-1.

Table 6-1: Traversing routes of five outbound trains and four inbound trains

5 Trains (Outbound)	Traversing Route	4 Trains (Inbound)	Traversing Route
T_1	A → H	T_6	H → A
T_2	I → H	T_7	H → A
T_3	A → H	T_8	H → I
T_4	A → H	T_9	H → A
T_5	I → H		

This train scheduling case can be transformed into a PMJSS problem, modelled by an improved activity-on-the-node disjunctive graph $DG = (\mathrm{O}, C, E)$, illustrated in Figure 6-2.

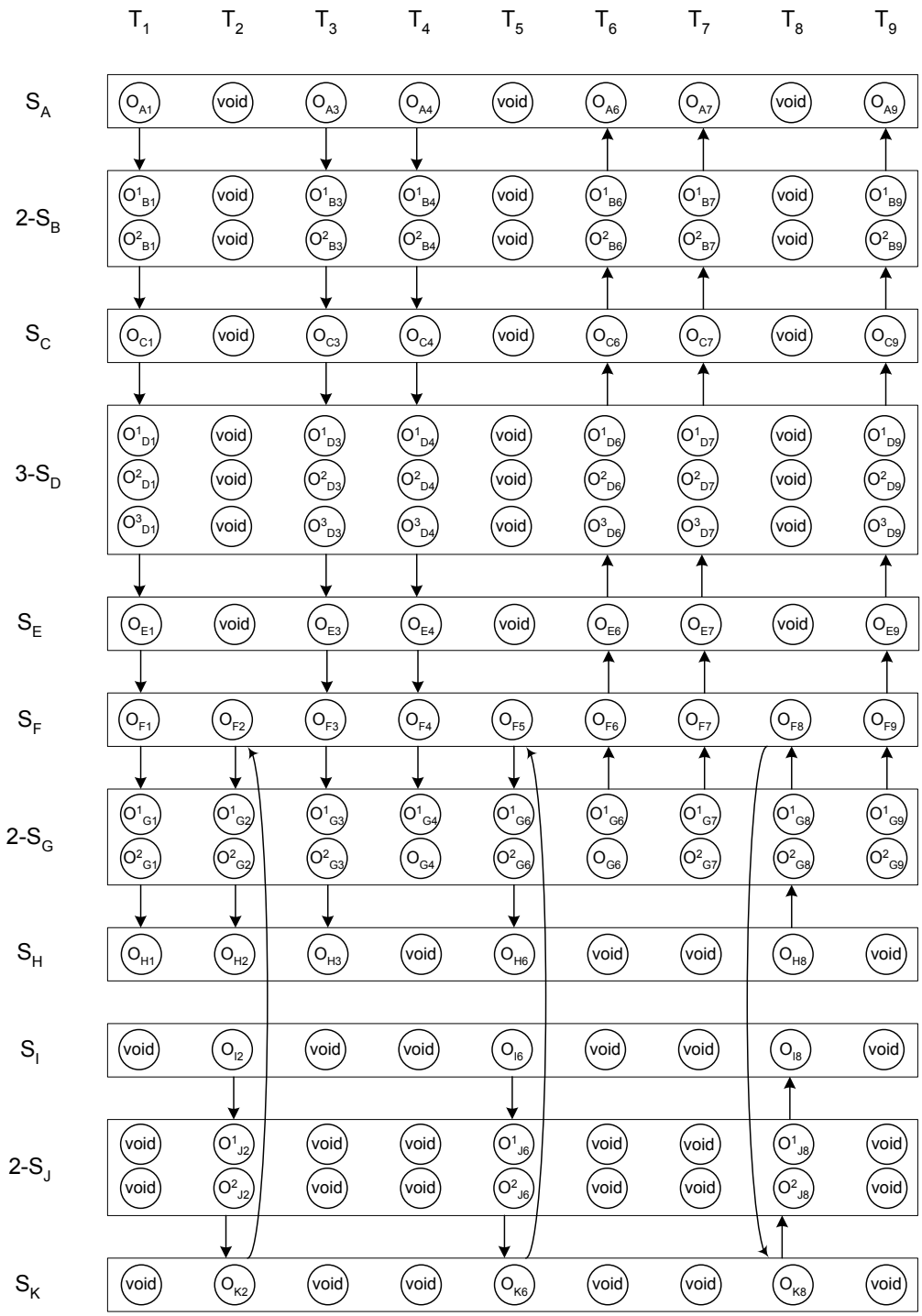

Figure 6-2: An improved disjunctive graph for PMJSS. Note that only conjunctive arcs are displayed in this graph for clarity.

The description for this improved *DG* is given as follows. Assuming that there are n jobs (trains) and m same-type machines (sections) in DG, we number the nodes

from 1 to N, where $N = n \times m$ denotes the total number of *actual* same-kind operations. For example, node O_{A1} represents an *actual* operation corresponding that Train 1 (T_1) traverses on Section A (S_A). If one train does not pass through a track section, the corresponding operation is defined as "*void*". Additionally, two *virtual* nodes (nodes F and L), which represent the start and the end of a schedule, are added respectively.

If one operation is *void*, its processing time is defined as zero. An arc (i, j) connects two *actual* nodes i and j, implying that operation i has to be processed immediately before operation j. Each arc has the length of p_i corresponding to the processing time of operation i.

In $DG = (O, C, E)$, C is the set of *conjunctive arcs* representing the fixed precedence relations between each pair of two operations belonging to the same job (train). For example, in Figure 6-2 for Train 1 (T_1) with prescribed traversing route from Section A to Section H, the set of conjunctive arcs for T_1 can be continuously listed as:

$$O_{A1} \rightarrow O_{B1} \rightarrow O_{C1} \rightarrow O_{D1} \rightarrow O_{E1} \rightarrow O_{F1} \rightarrow O_{G1} \rightarrow O_{H1}.$$

In addition, the directed arcs are added for connecting the virtual start (end) node with the first (last) operation of each train.

On the other hand, E is the set of *disjunctive arcs*, which represent undirected precedence relationships between each pair of two operations processed on the same machine (section). Actually, determining a PMJSS schedule corresponds with selecting one direction for each pair of disjunctive arcs and discarding the redundant arcs in E.

As a result, $DDG = (O, C, E')$ denotes the *directed disjunctive graph* representing one schedule. The length of a longest path from the virtual start node to the virtual end node in DDG is equivalent to the makespan of the schedule. A feasible schedule requires that DDG is acyclic and satisfies all the required constraints.

For the sake of figure's clarity, only the conjunctive arcs are drawn in Figure 6-2. To complete the graph model, for the above PMJSS instance, the rest of the arcs that need to be added are separately explained and analysed in the following paragraphs.

For each machine (section) in Figure 6-2, the nodes encircled by a rectangle represent the operations that are processed on the same-type machine (the same-type track section). For instance, on section A (S_A), the encircled actual nodes include O_{A1}, O_{A3}, O_{A4}, O_{A6}, O_{A7}, O_{A9}. Note that the set of disjunctive arcs connecting these encircled nodes are temporarily omitted in Figure 6-2 because of its complexity.

Table 6-2: The disjunctive arcs on a single-track section with six trains

Nodes	Disjunctive Arcs on a Single-Track Section
O_{A1}	$O_{A1} \rightarrow O_{A3}$; $O_{A1} \rightarrow O_{A4}$; $O_{A1} \rightarrow O_{A6}$; $O_{A1} \rightarrow O_{A7}$; $O_{A1} \rightarrow O_{A9}$;
O_{A3}	$O_{A3} \rightarrow O_{A1}$; $O_{A3} \rightarrow O_{A4}$; $O_{A3} \rightarrow O_{A6}$; $O_{A3} \rightarrow O_{A7}$; $O_{A3} \rightarrow O_{A9}$;
O_{A4}	$O_{A4} \rightarrow O_{A1}$; $O_{A4} \rightarrow O_{A3}$; $O_{A4} \rightarrow O_{A6}$; $O_{A4} \rightarrow O_{A7}$; $O_{A4} \rightarrow O_{A9}$;
O_{A6}	$O_{A6} \rightarrow O_{A1}$; $O_{A6} \rightarrow O_{A3}$; $O_{A6} \rightarrow O_{A4}$; $O_{A6} \rightarrow O_{A7}$; $O_{A6} \rightarrow O_{A9}$;
O_{A7}	$O_{A7} \rightarrow O_{A1}$; $O_{A7} \rightarrow O_{A3}$; $O_{A7} \rightarrow O_{A4}$; $O_{A7} \rightarrow O_{A6}$; $O_{A7} \rightarrow O_{A9}$;
O_{A9}	$O_{A9} \rightarrow O_{A1}$; $O_{A9} \rightarrow O_{A3}$; $O_{A9} \rightarrow O_{A4}$; $O_{A9} \rightarrow O_{A6}$; $O_{A9} \rightarrow O_{A7}$;

The number of disjunctive arcs on a single-track section (single machine) is equal to $n'_K(n'_K-1)$, where n'_K is the number of trains traversing on a single-track section S_K. For example, the subset of disjunctive arcs (E_A) when six trains (T_1, T_3, T_4, T_6, T_7, T_9) traverse on a single-track section (S_A) are enumerated in Table 6-2. The number of total disjunctive arcs in subset E_A is 30.

The characteristics of disjunctive arcs on a multiple-track section are much more complex. According to our analysis, the number of disjunctive arcs on a multiple-track section is equal to $n'_K(n'_K-1)u^2$, where u is the number of same-type section units (or parallel-machine units). For simplicity, we consider a double-track section S_J on which only three trains traverse. The total disjunctive arcs of subset E_J on S_J are listed in Table 6-3, where O^1_{J6} and O^2_{J6} are the same-type operation of train T_6

but they are processed on two different units of section S_J, i.e. Unit 1 and Unit 2. The number of total disjunctive arcs in subset E_J equals 24.

Table 6-3: The disjunctive arcs on a double-track section with three trains

Nodes	Disjunctive Arcs on a Double-Track Section
O^1_{J2}	$O^1_{J2} \rightarrow O^1_{J6}$; $O^1_{J2} \rightarrow O^1_{J8}$; $O^1_{J2} \rightarrow O^2_{J6}$; $O^1_{J2} \rightarrow O^2_{J8}$;
O^2_{J2}	$O^2_{J2} \rightarrow O^1_{J6}$; $O^2_{J2} \rightarrow O^1_{J8}$; $O^2_{J2} \rightarrow O^2_{J6}$; $O^2_{J2} \rightarrow O^2_{J8}$;
O^1_{J6}	$O^1_{J6} \rightarrow O^1_{J2}$; $O^1_{J6} \rightarrow O^1_{J8}$; $O^1_{J6} \rightarrow O^2_{J2}$; $O^1_{J6} \rightarrow O^2_{J8}$;
O^2_{J6}	$O^2_{J6} \rightarrow O^1_{J2}$; $O^2_{J6} \rightarrow O^1_{J8}$; $O^2_{J6} \rightarrow O^2_{J2}$; $O^2_{J6} \rightarrow O^2_{J8}$;
O^1_{J8}	$O^1_{J8} \rightarrow O^1_{J2}$; $O^1_{J8} \rightarrow O^1_{J6}$; $O^1_{J8} \rightarrow O^2_{J2}$; $O^1_{J8} \rightarrow O^2_{J6}$;
O^2_{J8}	$O^2_{J8} \rightarrow O^1_{J2}$; $O^2_{J8} \rightarrow O^1_{J6}$; $O^2_{J8} \rightarrow O^2_{J2}$; $O^2_{J8} \rightarrow O^2_{J6}$;

With the above analysis, the set of disjunctive arcs (E) can be partitioned into m subsets $E_1, E_2, \ldots\ldots, E_m$ where m is the number of same-type sections (machines). For example, E_k $(\forall k = 1,2,...,m)$ denotes the subset of disjunctive arcs connecting the encircled actual nodes on section k (machine k).

Let $E' = \bigcup E'_k (\forall k = 1,2,\ldots,m)$ be such a set of *directed* disjunctive arcs, where E'_k is the subset of directed disjunctive arcs on machine k. Let M_0 be the set of machines that have been sequenced, so at the start and the end there are $M_0 = \phi$ and $M_0 = M$, respectively. If $M - M_0 \neq \phi$, a selection of $S = \bigcup E'_k, \forall k \in M_0$ is called a partial selection; otherwise it is called a complete selection. Associated with a partial selection, there is one *partial directed disjunctive graph* denoted as $PDDG_S := (O, C \bigcup S) \mid S \subseteq E'$, associated with the selection S. If $PDDG_S$ is acyclic, S must be acyclic. However, even if S is acyclic, $PDDG_S$ may be still cyclic, implying that the schedule is infeasible. In a sense, a feasible PMJSS schedule can be represented as an acyclic directed disjunctive graph that corresponds to the unique complete selection S on both single machines and parallel machines.

With the above analysis based on the improved disjunctive graph model, an improved *shifting bottleneck procedure* (SBP) algorithm is developed to solve the PMJSS problem.

6.2.3 An Improved Shifting Bottleneck Procedure Algorithm

It is well known that the idea of *shifting bottleneck procedure* (SBP) algorithm is initiative as a solution to the *classical* job-shop scheduling (JSS) problem (Adams *et al.*, 1988). This is due to the analysis that a processing order of operations (i.e. job sequence) on machine k is equivalent to an acyclic selection E'_k that contains exactly either of each pair of disjunctive arcs in E_k. In a sense, after decomposing the JSS problem, sequencing the job sequence on a machine is similar to solving a set of *single-machine scheduling* (SMS) subproblems with different release dates and delivery times.

According to the above analysis, it is essential to build up a *partial disjunctive graph* $PDG = (\mathrm{O}, C, \bigcup E'_k \mid k \in M_0, \bigcup E_u \mid u \in M - M_0)$, where M_0 is the subset of machines that have been sequenced, $\bigcup E'_k \mid k \in M_0$ is such a subset of *directed* disjunctive arcs on sequenced machines, $\bigcup E_u \mid u \in M - M_0$ is the subset of disjunctive arcs on unsequenced machines.

Then, based on the constructed partial disjunctive graph PDG, the PMJSS problem is able to be decomposed as a set of *single-machine scheduling* (SMS) and/or *parallel-machine scheduling* (PMS) subproblems. The number of SMS and PMS subproblems is respectively equal to the number of unsequenced single-track sections (single machines) and unsequenced multiple-track sections (parallel machines) in PDG.

In this study, the original-version SBP algorithm for the classical JSS problem is greatly improved to solve the PMJSS problem in these four aspects:

- the topological-sequence algorithm (See Section 4.4.2) for solving the classical multi-stage scheduling problems is employed here to solve the

partial PMJSS graph model and then decompose the PMJSS problem into a set of SMS and/or PMS subproblems;

- a modified Carlier algorithm (See Section 6.2.9) based on the proposed lemmas is developed for solving the SMS subproblems;
- an extended Jackson algorithm (See Section 6.2.10) is implemented to solve the PMS subproblems;
- metaheuristic algorithm (See Section 4.4.4) is embedded under the architecture of SBP to re-optimise the partial sequence and to further optimise the complete sequence after obtaining the complete sequence.

The procedure of this improved-version SBP algorithm for the PMJSS problem is described as follows.

Step 1: Set $M_0 = \phi$. The initial *partial directed disjunctive graph* ($PDDG$) contains all the conjunctive arcs and no disjunctive arcs, i.e. $PDDG = (O, C)$.

Step 2: Do the following for each unsequenced machine $u \in M - M_0$:

Step 2.1: Implement the topological-sequence algorithm (See Section 4.4.2 for detail) to decompose the PMJSS problem into a set of SMS (or PMS) subproblems.

Step 2.2: Solve the SMS subproblems (See Section 6.2.9 for detail) by the modified Carlier algorithm.

Step 2.3: Solve the PMS subproblems by the extended Jackson algorithm (See Section 6.2.10 for detail).

Step 3: Bottleneck Selection and Sequencing:

Step 3.1: Choose machine k as bottleneck machine by $L_{\max}(k) = \max_{i \in M - M_0} (L_{\max}(i))$.

Step 3.2: Set the job-sequence on machine k according to the results obtained in *Step 2.2* or *2.3*.

Step 3.3: Update $M_0 = M_0 \bigcup \{k\}$.

Step 3.4: Update the partial directed disjunctive graph by adding E'_k, i.e. $PDDG = (O, C, \bigcup E'_k \mid k \in M_0)$

Step 4: Re-sequence and Re-optimise the partial directed disjunctive graph:

Step 4.1: If $M_0 \equiv M$, go to *Step 5*.

Step 4.2: Otherwise, optimise the current partial directed disjunctive graph by metaheuristics and then go to *Step 2*.

Step 5: Optimise the complete directed disjunctive graph:

Step 5.1: Optimise the *complete directed disjunctive graph* $CDDG = (O, C, \bigcup E'_k \mid k \in M)$ by metaheuristics (See Section 4.4.4 for details).

Step 5.2: If the solution is equal to the lower bound or known optimum, then stop; otherwise, run till the stopping conditions of metaheuristics are satisfied.

In a word, the SBP algorithm schedules machines consecutively, one at a time. At each SBP iteration, two key decisions have to be made in Step 2, namely, how to decompose the PMJSS problem into a set of SMS and/or PMS subproblems as well as how to solve these subproblems so that the bottleneck machine and its corresponding job sequence are effectively and efficiently determined.

6.2.4 Topological-Sequence Algorithm for Decomposing PMJSS

At each SBP iteration, we need to build up a partial directed disjunctive graph model, $PDDG = (\mathrm{O}, C, \bigcup E'_k \mid k \in M_0)$, which consist of the conjunctive arcs (i.e. C) of operations belonging to the same job and the directed disjunctive arcs (i.e. $\bigcup E'_k$) of operations processed on sequenced machines ($\forall k \in M_0$).

Based on the $PDDG$, we apply the topological-sequence algorithm to efficiently compute the values of heads e_j (starting times or the longest distance from the virtual start node to this node) and tails l_j (delivery times or the longest distance from this node to the virtual end node) for each node j ($\forall j \in \mathrm{O}$). The detailed procedure of this algorithm can be found in Section 4.4.2 or referred to Liu and Ong (2002).

6.2.5 Mathematical Formulation for Subproblems

After decomposing the PMJSS problems into a set of *single-machine scheduling* (SMS) and/or *parallel-machine scheduling* (PMS) subproblems, the key part of *shifting bottleneck procedure* is converted to solve the SMS and/or PMS subproblems with the different release times and delivery times. The objective of SMS and PMS subproblems is to minimise the maximum lateness.

The formulation of the SMS subproblem is described as below. Given a set of jobs $J = \{1, 2, ..., n\}$, each job j ($\forall j \in J$) has processing time p_j, release time r_j, starting time e_j and delivery time q_j. For a given schedule, the lateness L_j of job j is defined as $L_j = C_j - d_j$, that is, the (positive or negative) time difference between the completion time C_j and the due date d_j. The objective is to minimise the maximum lateness $L_{\max} = \max_{j \in J} L_j = \max_{j \in J}(C_j - d_j)$. Actually, for the SMS subproblem, minimising the maximum lateness is equivalent to minimising the makespan. The proof is given as below.

As each job j has a non-negative delivery time q_j (i.e. the longest distance from node j to the virtual last node), the completion-delivery time is defined as $f_j = C_j + q_j$. For the SMS subproblem with the different release times and delivery times, the makespan is equal to the maximum completion-delivery time. The equivalence between two objectives can easily be demonstrated by subtracting a sufficiently large constant, that is,

- $const = C_{\max}$ ($C_{\max}$ is the makespan of the *partial directed disjunctive graph* of the PMJSS problem)
- $\because d_j = C_{\max} - q_j = const - q_j$
- $\therefore L_j = C_j - d_j = C_j - (const - q_j) = C_j + q_j - const = f_j - const$
- $\therefore L_{\max} = \max_{j \in J} L_j = \max_{j \in J}(f_j - const)$
- $\therefore \min L_{\max} \cong \min f_{\max}$

Therefore, decomposed from the PMJSS problem, a SMS subproblem related to one machine k ($\forall k \in M - M_0$) and a current set of sequenced machines M_0 is mathematically formulated as follows.

SMS Model: $SMS(k, M_0)$

Minimise $C_{\max}^k \ (\forall k \in M - M_0)$

The objective function is to minimise the makespan of the SMS subproblem.

Subject to:

$$C_{\max}^k \geq e_i + p_i + q_i, \ \forall i \in \mathrm{O}_k \tag{6.9}$$

Equation (6.9) restricts that the completion times of the operations processed on machine k are no earlier than the makespan.

$$e_i \geq e_j + p_j \vee e_j \geq e_i + p_i, \ \forall (i, j) \in E_k \tag{6.10}$$

Equation (6.10) defines the precedence relationships of jobs i and j processed on machine k.

$$e_i \geq r_i, \ \forall i \in \mathrm{O}_k \tag{6.11}$$

Equation (6.11) satisfies the job-ready conditions.

Similarly, one PMS subproblem decomposed from the PMJSS problem can be mathematically modelled as below.

PMS Model: $PMS(k, M_0)$

Minimise $C_{\max}^k \ (\forall k \in M - M_0)$

The objective function is to minimise the makespan of the PMS subproblem.

Subject to:

$$C_{\max}^k \geq t_i + p_i + q_i, \ \forall i \in \mathrm{O}_k \tag{6.12}$$

$$t_j \geq t_i + p_i + L(y_{ijz} - 1), \ \forall (i, j) \in E_k \tag{6.13}$$

$$t_i \geq t_j + p_j + L(y_{jiz} - 1), \ \forall (i, j) \in E_k \tag{6.14}$$

$$y_{ijz} + y_{jiz} \leq 1, \ \forall (i, j) \in E_k \tag{6.15}$$

$$x_{iz} + x_{jz} - 1 \leq y_{ijz} + y_{jiz}, \ \forall (i, j) \in E_k \tag{6.16}$$

$$\sum\nolimits_z x_{iz} = 1, \ \forall i \in N_k \tag{6.17}$$

$$y_{ijz}, y_{jiz}, x_{iz}, x_{jz} = \begin{cases} 0 \\ 1 \end{cases}, \ \forall (i,j) \in E_k \tag{6.18}$$

$$t_i \geq r_i, \ \forall i \in \mathrm{O}_k \tag{6.19}$$

where $x_{iz} = 1$ if operation i is assigned to the z^{th} unit of parallel-machine k, otherwise $x_{iz} = 0$; $y_{ijz} = 1$ if both operation i and j are assigned to the z^{th} unit of parallel-machine k and operation i precedes operation j, $y_{ijz} = 0$ otherwise; and L is a sufficiently large number. In PMS model, equations (6.13-18) particularity restricts that the precedence relationship between a pair of operations processed on a unit of parallel machine is exclusive and each unit can process at most one job at a time.

6.2.6 Schrage Algorithm

As discussed in Chapter 3, for the *basic* single-machine scheduling (SMS) problem in which all the release times are set as zero, the maximum lateness can be optimally solved by the *earliest due date* (EDD) dispatching rule.

However, in the existence of different release dates and delivery times, the SMS problem becomes much more complex.

Actually, the SMS problem with different release dates and delivery times can be modelled as a three-machine *flow-shop scheduling* (FSS) problem which is NP-hard. In this context, the release dates are often regarded as *heads*, the processing times as *bodies*, and the delivery times as *tails*. Jobs have to be processed on machines 1, 2 and 3 in order. In other words, job j spends time r_j (the head) on machine 1, p_j (the body) on machine 2, and q_j (the tail) on machine 3.

In the literature, most of the approximation algorithms for solving the SMS problem are based on variations of the *extended EDD* rule, which is briefly called the *Schrage algorithm*, since the extended *EDD* rule was introduced by Schrage (1971). The procedure of the Schrage algorithm for solving the SMS subproblem is given as follows.

Step 1: Set a time point, $t = \min_{i \in I} r_i$. Set $U = \phi$, and $\bar{U} = J$, where U is the set of scheduled jobs; $\bar{U} = J - U$ is the set of unscheduled jobs;

Step 2: At time point t, if $j \in \bar{U}$ and $r_j \le t$ and $q_j = \max_{i \in \bar{U}} q_i$, schedule job j.

Step 3: Then, update $U = U \bigcup \{j\}$ and $\bar{U} = \bar{U} - \{j\}$. Set the starting time of job j, $e_j = t$. Reset $t = \max(t_j + p_j, \min_{i \in \bar{U}} r_i)$. If $\bar{U}$ is empty, stop; otherwise, go to Step 2;

For illustrating the Schrage algorithm, a numerical example is provided in Appendix 4.

6.2.7 Carlier Algorithm

The properties of a Schrage schedule (i.e. the schedule obtained by the Schrage algorithm) are analysed in the following:

1) The sequence of jobs $a, a+1, ..., c$ forms a block in a Schrage schedule and is called the ***critical sequence*** with the corresponding set $\Lambda = \{a, a+1, ..., c\}$,. if $C_{\max} = e_i + p_i + q_i, \forall i \in \Lambda$.
2) Job c is called the ***critical job*** because it attains the makespan in the Schrage schedule.
3) Job a is called the ***idled job***, i.e. the first job in critical sequence Λ. Thus, there may be idle time before a as there must not be idle time between the processing of jobs a and c.
4) It is obvious for $r_i \ge r_a$, $\forall i \in \Lambda$.
5) The job satisfying the property $q_w < q_c$ and having the largest subscript in critical sequence is called the ***interference job*** w.

Based on the above analysis, Carlier (1982) proposed two theorems for the Schrage schedule.

Theorem 1:

Let $C_{Schrage}$ be the makespan of the Schrage schedule. If the Schrage schedule is not optimal, there is a interference job w and a critical subset $\Lambda_w = \{w+1,...,c\}$ out of critical sequence $\Lambda = \{a, a+1,...,c\}$ such that,

$$h(\Lambda_w) = \min_{j\in\Lambda_w} r_j + \sum_{j\in\Lambda_w} p_j + \min_{j\in\Lambda_w} q_j > C_{Schrage} - p_w \tag{6.20}$$

Thus, the distance from the makespan of Schrage schedule to the optimal makespan is less than p_w. Moreover, in an optimal schedule, job w will be processed either before or after all the jobs of critical subset $\Lambda_w = \{w+1,...,c\}$.

Theorem 2:

If this Schrage schedule is optimal, there exists Λ_w such that $h(\Lambda_w) = C_{Schrage}$.

Carlier Algorithm is a branch and bound method which is based on the Schrage Algorithm, the interference job w and a critical subset Λ_w. The description of one branching tree is given in Figure 6-3.

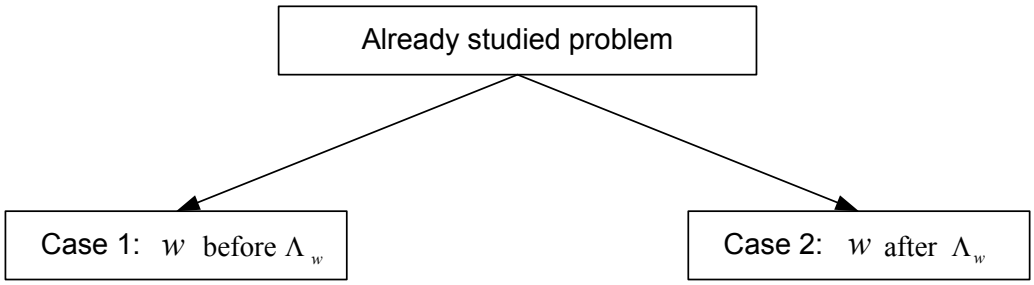

Figure 6-3: The Branch and Bound Scheme of Carlier Algorithm.

The node of the tree with the lowest bound is considered and the Schrage algorithm for the SMS subproblem is associated with it. If interference job w does not exist, the schedule is optimal according to Carlier theorems; otherwise, the distance to the optimum is less than the processing time of interference job w, p_w. Thus, either job w will be processed before all the jobs of Λ_w, or after all the jobs of Λ_w.

In case 1, interference job w will be processed before Λ_w, by resetting the delivery time of job w as

$$q_w = \max\{q_w, \sum_{j\in\Lambda_w} p_j + q_c\} \tag{6.21}$$

In case 2, interference job w will be processed after Λ_w by resetting the release time of job w as

$$r_w = \max\{r_w, \min_{j\in\Lambda_w} r_j + \sum_{j\in\Lambda_w} p_j\} \tag{6.22}$$

However, many researchers found that Theorem 2 proposed by Carlier (1982) may be incorrect, which can be simply proved by one enumerative example given as below. The data of a two-job SMS instance is given in Table 6-4.

Table 6-4: The data of a two-job SMS instance

Job i	1	2
r_i	10	12
p_i	3	3
q_i	11	12

The directed disjunctive graph and Gantt chart for this SMS example is drawn in Figure 6-4.

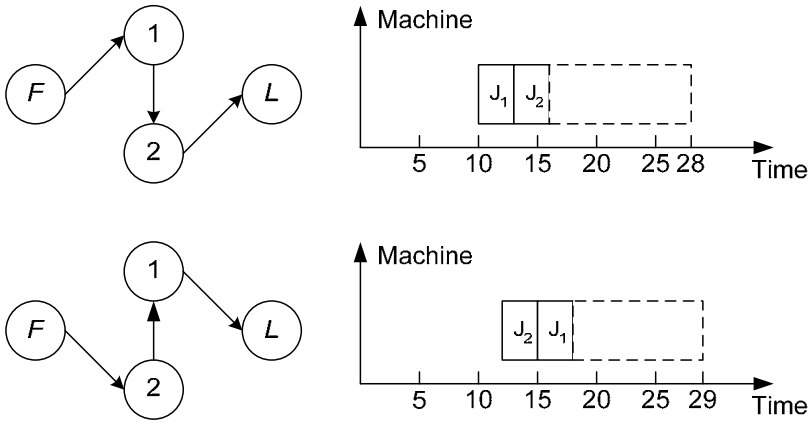

Figure 6-4: The directed disjunctive graph and Gantt chart for a two-job SMS instance

From the above figure, it is known that the optimal makespan is 28 and the optimal SMS schedule is $\{J_1, J_2\}$.

If $\{J_1, J_2\}$ is also a Schrage schedule, then we have a critical path $\{F, a, a+1, ..., c, L\}$ = $\{F, 1, 2, L\}$, critical sequence Λ = $\{1, 2\}$, interference job w = 1 as $q_1 < q_2$, critical subset $\Lambda_w = \{2\}$, and the makespan equal to 28. According to

$h(\Lambda_w) = \min_{j\in\Lambda_w} r_j + \sum_{j\in\Lambda_w} p_j + \min_{j\in\Lambda_w} q_j$ used in Carlier's Theorem 1, the value of $h(\Lambda_w)$ for this Schrage schedule is calculated as below:

$$h(\Lambda_w) = h(\{2\}) = r_2 + p_2 + q_2 = 12 + 3 + 12 = 27$$

As this Schrage schedule is optimal, it is contradicted to Carlier's Theorem 2 because $h(\Lambda_w)$ equals 27 and the makespan ($C_{\max}$) of this schedule is 28, i.e. $h(\Lambda_w) \neq C_{\max}$ in this case.

Moreover, for all possible sets of jobs (i.e. {2}, {1}, {1,2},{2,1}) , we obtain

$$h(\Lambda_w) = h(\{2\}) = r_2 + p_2 + q_2 = 12 + 3 + 12 = 27$$

$$h(\Lambda_w) = h(\{1\}) = r_1 + p_1 + q_1 = 10 + 3 + 11 = 24$$

$$h(\Lambda_w) = h(\{1,2\}) = r_1 + p_1 + p_2 + q_1 = 10 + 3 + 3 + 11 = 27$$

$$h(\Lambda_w) = h(\{2,1\}) = r_1 + p_1 + p_2 + q_1 = 10 + 3 + 3 + 11 = 27$$

Obviously, there is no possibility of having a subset of jobs to satisfy Theorem 2 proposed by Carlier (1982). Further computational experiments in the next section can also demonstrate that this theorem may be wrong.

6.2.8 Five Proposed Lemmas

Based on our own analysis, five lemmas are proposed and the proofs for the proposed lemmas are correspondingly given.

Lemma I:

For a Schrage schedule and Λ is a critical sequence in this Schrage schedule, there exists $C_{\max} = LB(\Lambda) + q_c - \min_{j\in\Lambda} q_j$

Lemma II:

If $q_c = \min_{j\in\Lambda} q_j$, then $C_{\max} = LB(\Lambda)$.

Proof for Lemmas I and II:

For any subset $A \subseteq J$ of jobs, the inequality is certain,

$$Opt \geq LB(A)$$

$$LB(A) = \min_{j \in A} r_j + \sum_{j \in A} p_j + \min_{j \in A} q_j .$$

If $A = \Lambda$ and $\Lambda = \{a, a+1, ..., c\}$, $r_a = \min_{j \in \Lambda} r_j$ and there is no idle time between the processing of jobs a and c, we obtain,

$$LB(\Lambda) = r_a + \sum_{j=a}^{c} p_j + \min_{j \in \Lambda} q_j .$$

In addition, it is certain that,

$$C_{\max} = e_a + \sum_{j=a}^{c} p_j + q_c$$

where e_a is the starting time of job a.

As $e_a = r_a$, we obtain,

$$C_{\max} = r_a + \sum_{j=a}^{c} p_j + q_c .$$

Therefore, $C_{\max} = LB(\Lambda) + q_c - \min_{j \in \Lambda} q_j$.

Lemma III:

If there is an interference job w, then $C_{\max} - LB(\Lambda_w) < p_w$.

Proof for Lemma III:

Lemma III is basically similar to Theorem 1 proposed by Carlier. To complete analysing the properties of the Schrage schedule, here we give our own demonstration which is different from the proof proposed by Carlier (1982).

As $\Lambda_w = \{w+1, ..., c\}$ is the subset of jobs from the critical sequence which are processed after the interference job w, it is clear that $q_w < q_c \leq q_j$ and $e_w < r_j$ hold for all $j \in \Lambda_w$, where e_w is the starting time of the interference job w. So, inequality applied to $\Lambda_w = \{w+1, ..., c\}$ generates another lower bound:

$$LB(\Lambda_w) = \min_{j \in \Lambda_w} r_j + \sum_{j \in \Lambda_w} p_j + \min_{j \in \Lambda_w} q_j > e_w + \sum_{j \in \Lambda_w} p_j + q_c .$$

Since there is no idle time during the execution of the jobs in the critical sequence, the makespan of the Schrage schedule is:

$$C_{\max} = e_w + p_w + \sum_{j \in \Lambda_w} p_j + q_c .$$

Because $LB(\Lambda_w) > e_w + \sum_{j \in \Lambda_w} p_j + q_c$, we obtain

$$C_{\max} - LB(\Lambda_w) < C_{\max} - (e_w + \sum_{j \in \Lambda_w} p_j + q_c)$$

Therefore, it is proved that $C_{\max} - LB(\Lambda_w) < p_w$.

Lemma IV:

For a Schrage schedule, $C_{\max} = \min_{j \in \Lambda} e_j + \sum_{j=a}^{c} p_j + \min_{j \in \Lambda} q_j$ exists.

Lemma V:

Even if $q_c = \min_{j \in \Lambda} q_j$, this Schrage schedule may not be optimal.

The proofs for Lemmas IV and V are observed and induced by solving many numerical examples, including the SMS test problem provided by Carlier (1982). Using this test problem, the demonstration procedure is given in Appendix 5.

6.2.9 A Modified Carlier Algorithm

Lemma V proposed above indicates that the stopping condition (i.e. "*Schrage schedule is optimal if* $q_c = \min_{j \in \Lambda} q_j$, $\forall j \in \Lambda$") introduced in the book (Chapter 10-5, Leung 2004) may be incorrect or incomplete. As a result, the question generated is: "what is the appropriate stopping condition that can be implemented for the Carlier algorithm?". Therefore, with answering this question, a modified-version Carlier algorithm is developed for solving the SMS subproblem.

The procedure of the proposed modified Carlier algorithm is described as follows:

Step 1: Apply Schrage algorithm to the current SMS problem and save the Schrage results.

Step 2: If there exist the interference job w satisfying $q_w < q_c$, the interference job w will be processed after critical subset Λ_w by setting $r_w = \max\{r_w, \min_{j\in\Lambda_w} r_j + \sum_{j\in\Lambda_w} p_j\}$ and then go to Step 1. Otherwise, choose the best Schrage schedule with minimum makespan value among the saved Schrage results.

6.2.10 An Extended Jackson Algorithm

In the literature, the basic *parallel-machine-scheduling* (PMS) problem was efficiently solved by a constructive heuristic based on Jackson's rule (1955). In the original form, Jackson's rule sequentially schedules the available job with the smallest due date. Carlier (1987) proposed a branch and bound algorithm to solve the PMS problem by extending Jackson's rule to schedule the job with the largest tail time instead of the smallest due date. Because the branch and bound algorithm is very time consuming and the suboptimal solution by constructive heuristic is good enough, here we extend Jackson's rule to quickly solve the PMS subproblems with release times and delivery times, at each iteration in our improved SBP algorithm for the PMJSS problem.

The procedure of the extended Jackson Algorithm for the PMS subproblem is described as follows.

Step 1: Initialisation: set the available times of machine units as zero; set the list of scheduled jobs as empty.

Step 2: While the list of scheduled jobs is not complete:

Step 2.1: Determine the parallel-machine unit that leads to the minimum available time; and break the tie arbitrarily.

Step 2.2: If the release times of all jobs in the list of unscheduled jobs are larger than the minimum available time of the chosen parallel-machine unit, then select the job having the minimum release time.

Otherwise, select the job having the longest delivery time among the jobs of which the release time is less than the minimum available time.

Step 2.3: Update the results according to the selected job: 1) add the selected job to the list of scheduled jobs; 2) erase the selected job from the list of unscheduled jobs; 3) set the starting time of the selected job; 4) set the new available time of the chosen parallel-machine unit.

6.2.11 Embedded Metaheuristics

As discussed in Chapter 4, the approaches of neighbourhood search guided by metaheuristics such as *Tabu Search* or *Simulated Annealing* offer a comparatively simple implementation but are very effective and efficient to solve the large-size problems with short computational time and good accuracy. Hence, to exploit the merits of metaheuristics, it is decided that *shifting bottleneck procedure* (SBP) and *metaheuristics* are combined as a hybrid algorithm for the PMJSS problem. Under the architecture of this improved-version SBP algorithm, metaheuristics are embedded for re-optimising the partial (and complete) PMJSS schedule. For details of metaheuristic methodology refer to Section 4.4.4 or four papers (Liu and Ong 2002; Liu and Ong; 2004; Liu *et al.* 2005a; Liu *et al.* 2005b).

6.2.12 Computational Experiments

The proposed hybrid SBP (HSBP) algorithm combined with TS has been coded in Visual C++ and tested on a HP desktop with Pentium IV 3.20 GHz Processor and 2 GB RAM. The computational experiment is to evaluate the effectiveness of the proposed HSBP algorithm and examine the improvement in solution quality when the TS algorithm is embedded under the framework of SBP.

First, the proposed HSBP algorithm without embedding the Tabu Search algorithm is tested on the benchmark JSS data collected by OR-Library

(http://people.brunel.ac.uk/~mastjjb/jeb/orlib/). The computational results for these problem instances (LA01-40) are summarised in Table 6-5, in which the deviation of one instance is calculated as the percentage of the obtained makespan away from the optimal value (*Opt*), namely, *Deviation* = 100×(*Makespan* – *Opt*)/*Opt*. The optimal value (*Opt*) is equal to the lower bound (*LB*) value or the best upper bound (*UB*) value known in the literature.

Table 6-5: Computational results by the proposed SBP algorithm without TS

JSS Instances	*n*	*m*	Average Deviation (%)	Average Time (s)
La01-05	10	5	2.08	0.156
La06-10	15	5	0.39	0.281
La11-15	20	5	0.00	0.453
La16-20	10	10	10.13	0.829
La21-25	15	10	7.98	1.672
La26-30	20	10	6.03	2.734
La31-35	30	10	0.70	5.746
La36-40	15	15	15.48	4.961

As shown in Table 6-5, the optimal solutions (most of them equal to the *LB*) of the instances La06-10, La11-15, and La31-35 can be easily found by the proposed HSBP algorithm because the number of jobs is several times larger than the number of machines (i.e. $n/m > 2$). For the other tough instances, the deviation becomes larger and larger when the problem size of the JSS instance increases, implying that the proposed HSBP algorithm without embedding combining the TS algorithm seems not able to meet the practical demand for high solution quality.

To verify the superiority of the proposed HSBP algorithm combined with the TS algorithm, the benchmark instances of La01-40 are further computed after activating the embedded components of the TS algorithm. Table 6-6 shows the comparison of the computational outcomes between the SBP algorithm without TS and the HSBP algorithm with TS. The comparison indicates that the HSBP with TS performs better than the pure SBP, especially for the large-size tough JSS instances. On the other hand, the HSBP algorithm with TS has to spend more running time

due to the fact that more computing efforts are made to guarantee to find the better or optimal solutions.

Table 6-6: Comparison of the computational outcomes between SBP and HSBP

JSS Instances	*n*	*m*	Average Deviation (%)		Average Time (s)	
			SBP	HSBP	SBP	HSBP
La01-05	10	5	2.08	0.00	0.156	3.716
La06-10	15	5	0.39	0.00	0.281	13.788
La11-15	20	5	0.00	0.00	0.453	31.286
La16-20	10	10	10.13	0.19	0.829	116.146
La21-25	15	10	7.98	0.36	1.672	328.125
La26-30	20	10	6.03	0.02	2.734	687.682
La31-35	30	10	0.70	0.00	5.746	184.807
La36-40	15	15	15.48	0.84	4.961	829.268

The solution quality of PMJSS is mainly dependent on the definition of the multiple-track sections (or parallel-machines) and the sequence of the so-called bottleneck section in which it has the minimum gaps (idle times) between trains. The computation results of PMJSS will be given in detail in Chapter 7.

6.3 Blocking Parallel-Machine Job-Shop Scheduling

Due to the lack of buffer or storage space, the real-world train scheduling problem should consider blocking or hold-while-wait constraints, which means that a track section cannot release and must hold the train until next section on the routing becomes available. In order to consider the blocking conditions in process, the train scheduling problem should be modelled as a *Blocking Parallel-Machine Job-Shop Scheduling* (BPMJSS) problem. This is also achieved by considering the train trips as jobs, which will be scheduled on single-track sections that are regarded as single machines, and on multiple-track sections that are referred to as parallel machines.

6.3.1 Mathematical Formulation for BPMJSS

Based on the mathematical programming for PMJSS in Section 6.2.1, the mathematical formulation for BPMJSS is completed by adding the blocking constraints defined by Eq. (6.23).

BPMJSS Model

Minimise $C_{\max}$

The objective function is to minimise the makespan.

Subject to:

……. (6.2-6.8)

$$\sum_{j=1}^{n}\sum_{l=1}^{h_k}\sum_{k=1}^{m} r_{jolk} s_{jlk} w_{ijolk} \geq \sum_{l=1}^{h_k}\sum_{k=1}^{m} r_{i,o+1,l,k} s_{ilk} \ , \ \forall i, i \neq j; \ o = 1, 2, \ldots, m-1. \quad (6.23)$$

In addition to constraints defined by Eqs. (6.2-6.8) for the PMJSS model in Section 6.2.1, Eq. (23) defines the blocking constraints especially for the BPMJSS model. To satisfy the blocking constraints, for each operation, the starting time of the same-machine successor should be greater or equal to the starting time of the same-job successor.

6.3.2 Feasibility Analysis for BPMJSS

As discussed in Chapter 5, most research works in *classical* scheduling problems are based on the disjunctive graph. However, a strong limitation that still remains in this classical disjunctive graph is that it disregards the capacity of intermediate buffers between machines. In fact, in many real-life situations especially for train scheduling, the inter-machine buffer capacity should be carefully taken into account. To incorporate this restriction, the alternative graph is proposed to model the *non-classical* scheduling problems with various inter-machine buffer conditions.

As the *blocking parallel-machine job-shop scheduling* (BPMJSS) problem is one type of *non-classical* scheduling problem, the alternative graph is employed to model and analyse the BPMJSS problem.

In the alternative graph, two types of operations are distinguished, namely *ideal* and *blocking*. An *ideal* operation, completed on a machine from its starting time to its completion time, at once leaves this machine which becomes immediately available for processing other operations. On the contrary, a *blocking* operation may remain on a machine even after its completion time, thus blocking this machine.

For illustration, we analyse the characteristics of alternative arcs with four related operations, in comparison with disjunctive arcs.

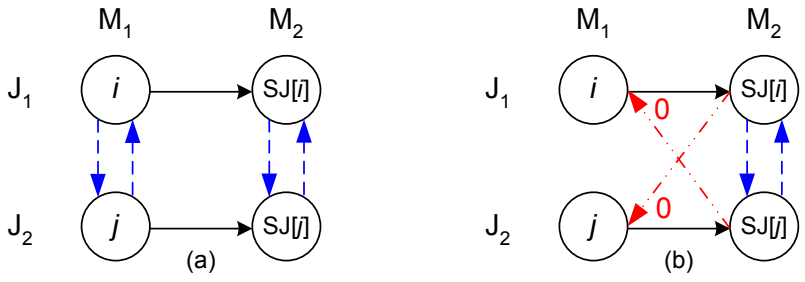

Figure 6-5: (a) Analysis of disjunctive arcs; and (b) analysis of alternative arcs.

In Figure 6-5(a), disjunctive arcs are drawn if all of these four operations are *ideal*. Due to the lack of buffer storage between machines M_1 and M_2, operation i (or operation $j = SM[i]$ that is the same-machine successor of operation i) may block machine M_1 if the downstream machine M_2 is not available, then one pair of

alternative arcs (i.e. $((SJ[i] \to j),(SJ[j] \to i))$ shown in Figure 6-5(b)) with the weight of zero is used to replace a pair of disjunctive arcs (i.e. $((i \to j),(j \to i))$ shown in Figure 6.6(a)). In this case, we say that $SJ[i] \to j$ is the alternative of $(SJ[j] \to i)$ or $(SJ[j] \to i)$ is the alternative of $SJ[i] \to j$. In this definition, an alternative arc belongs to one pair only. Given a pair of alterative arcs $((SJ[i] \to j),(SJ[j] \to i))$, we say that $SJ[i] \to j$ is selected whereas $SJ[i] \to j$ should be forbidden, and vice versa.

Thus, as drawn in Figure 6-6, the directed disjunctive graph and the directed alternative graph are respectively illustrated by the Gantt chart if the selections between each pair of disjunctive arcs or alternative arcs have been made.

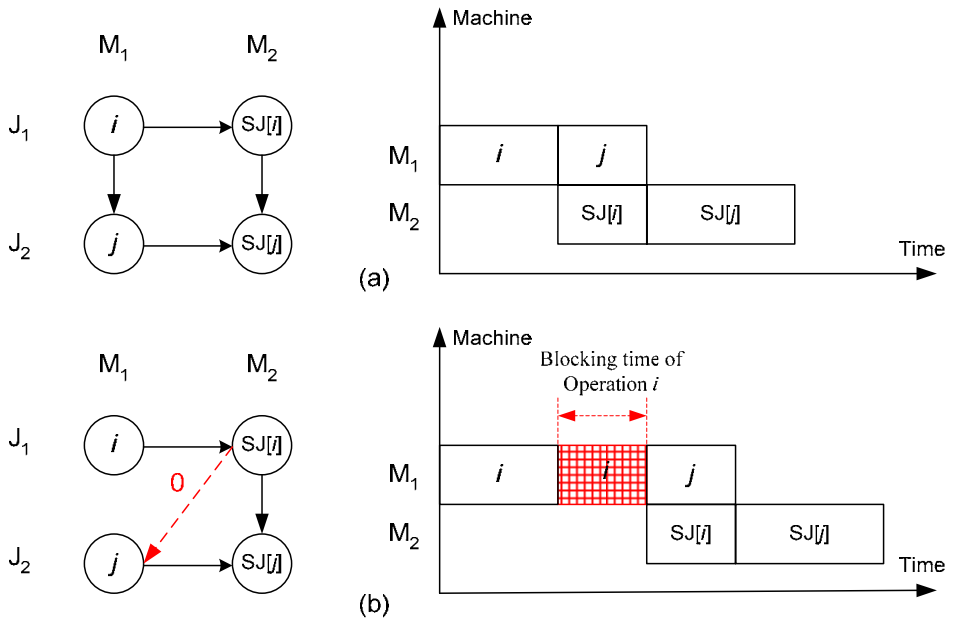

Figure 6-6: Gantt chart for illustrating the directed disjunctive graph and the directed alternative graph.

In Figure 6-6(b), operation j cannot be processed immediately after operation i until machine M_1 is released when operation $SJ[i]$ can start to be processed on the downstream machine M_2. This is because that operation i has to block machine M_1 due to the lack of buffer storage.

Based on the above analysis, to satisfy the blocking constraints, we may have to insert the alternative arc (i.e. $SJ[Oper] \to SM[Oper]$) with zero weight for each

blocking operation in the alternative graph model. Unfortunately, this unavoidably causes the graph model to become cyclic (infeasible).

For an example shown in Figure 6-7(d), a cycle of operations is generated in the directed alternative graph, i.e., $O_1 \to O_2 \to O_3 \to O_5 \to O_4 \to O_1$, implying this schedule is infeasible.

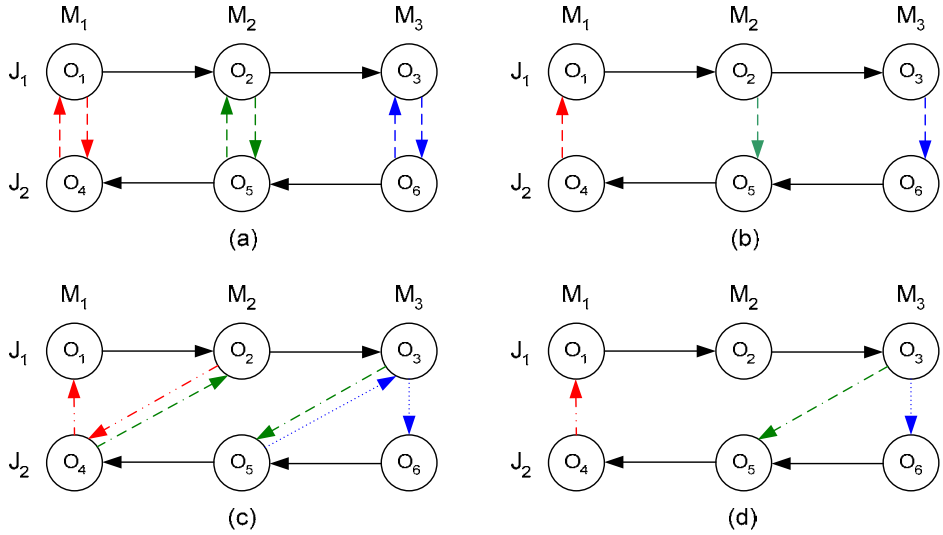

Figure 6-7: Analysing infeasibility occurrence: (a) disjunctive graph; (b) directed disjunctive graph; (c) alternative graph; and (d) directed alternative graph that is infeasible.

This phenomenon indicates that infeasibility occurs often in the directed alternative graph model. As a consequence, it is not easy to directly determine a feasible *non-classical* schedule with blocking conditions in job-shop environments.

Furthermore, blocking conditions can be distinguished into two cases: *swap-allowed blocking* and *no-swap blocking*. In a case where all jobs in the cycle must move simultaneously to the next machine in a cycle, this blocking situation is similar to "***deadlock***", as analysed in Section 5.4.4.

To be feasible, a "***swap***" manipulation is needed whenever there is a cycle of two or more jobs, each one waiting for a machine which is blocked by another job in the cycle. It is intuitive that, depending on the particular context, a swap may be allowed or not. The swap is allowed when the jobs can move independently of each other. On the contrary, the swap is not allowed when the jobs can move strictly after the subsequent resource becomes available. In a sense, the deadlock situation

is similar to a conflict when one outbound train and one inbound train are crossing in the single-track section. For safety, the deadlock situation is strictly prohibited in the train scheduling problems, namely, the train schedule should be deadlock-free. As the swap is not allowed for blocking trains, the deadlock-free conditions may be guaranteed only when some resources are available in multiple units (i.e. parallel machines or multiple-track sections). The deadlock and deadlock-free situations are respectively illustrated in Figure 6-8.

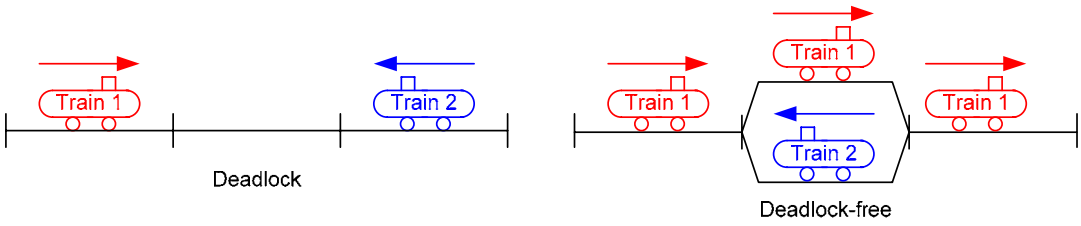

Figure 6-8: Illustration of deadlock and deadlock-free situations in train scheduling.

If the single machine (or single-track section) M_2 in Figure 6-7 is changed to be the parallel machine with two units (or double-track section), the deadlock-free status can be guaranteed with such a schedule shown in Figure 6-9.

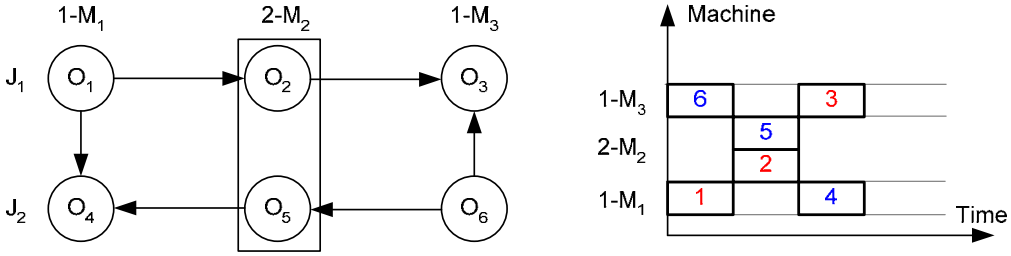

Figure 6-9: A deadlock-free schedule is achieved when machine M_2 is available in multiple units.

6.3.3 Feasibility-Satisfaction Procedure for BPMJSS

It is well known that the idea of *shifting bottleneck procedure* (SBP) is initiative as a solution to the classical JSS problem (Adams *et al.*, 1988). This is due to the analysis that a processing order of operations (i.e. job sequence) on one machine is equivalent to an acyclic selection of disjunctive arcs that contains exactly either of each pair of disjunctive arcs on this machine. Consequently, the machines can be

considered one by one and the results are applied to both rank the machines and select a good job sequence on the highest ranked machine (i.e. bottleneck machine).

First, without considering blocking conditions, a *parallel-machine job-shop-scheduling* (PMJSS) problem is solved by an improved-version SBP algorithm that has been presented in Section 6.2. Inspired by SBP algorithm for PMJSS, in order to satisfy blocking conditions and construct a feasible BPMJSS schedule, we then propose an innovative algorithm for BPMJSS. This algorithm is called *Feasibility Satisfaction Procedure* (FSP) which can schedule trains consecutively one at a time. The architecture of the proposed FSP algorithm for BPMJSS is described below.

- Apply developed SBP algorithm to solve the PMJSS model.
- Check whether the obtained schedule by SBP is feasible, i.e. satisfying all the blocking constraints.
- If feasible, add the new train and update the input data; apply SBP algorithm again to obtain new schedule and check feasibility again.
- If infeasible, load the feasible graph model generated at previous (or initial) step; apply the *Train-Insertion Algorithm* (See below) to insert the operations of the added train respectively to each section and satisfy the blocking constraints; then check whether this new graph model is feasible.
- Apply "Feasibility Satisfaction Procedure" (FSP) iterations till the feasible complete graph model is obtained.

In the above procedure, we design a *Train-Insertion Algorithm* which can effectively determine the new conflict-free timetable after inserting a new train into the current schedule. In this procedure, it is very important to construct an efficient data structure that can conveniently track all the necessary information for analysing and satisfying blocking conditions, including the direction of trains, sectional running times of each train, the earliest available time of single-track and/or multiple-track sections, the ready time, the starting time, the completion time, the blocking time, the departure time of inserted train's operation, etc. More specially, given a partial feasible schedule u^k at iteration k of Feasibility Satisfaction Procedure (FSP) and one train T_{k+1} to be added, the procedure of *Train-Insertion Algorithm* is described as follows.

Step 1: Let s denote a section and operation $O_{T_{k+1},s}$ denote the traversal of train T_{k+1} on s.

Step 2: If s is a single-track section, then go to Step 3; else if s is a multiple-track section, then go to *Step 4*.

Step 3: Assume that the train list of u^k on s is $(O_{T_1,s}, O_{T_2,s}, \ldots, O_{T_k,s})$, to find the best insertion point while satisfying the blocking conditions, we consider the following three choices in order:

i) whether $O_{T_{k+1},s}$ can be inserted before $O_{T_1,s}$;

ii) whether $O_{T_{k+1},s}$ can be inserted in the middle between $O_{T_1,s}$ and $O_{T_k,s}$ as early as possible; and

iii) otherwise, insert $O_{T_{k+1},s}$ after $O_{T_k,s}$.

Step 4: As s is a multiple-track section, assume the number of total track units is n_s and $s = \{s_{u_1}, s_{u_2}, \ldots, s_{u_{n_s}}\}$. In addition, each unit is associated with the train's direction i.e. $s = \{s_{u_1}, s_{u_2}, \ldots, s_{u_{n_s}}\} = \{s_{inbound}\} \oplus \{s_{outbound}\}$. If the direction of train T_{k+1} is inbound, consider inserting $O_{T_{k+1},s}$ on one unit of a multiple-track section s_{u_i} ($s_{u_i} \in \{s_{inbound}\}$) having the earliest available time. In this way, s_{u_i} is treated as a single-track section. If the direction of train T_{k+1} is outbound, it is in the same fashion but with $s_{u_i} \in \{s_{outbound}\}$.

Step 5: Go to *Step 1* for the next adjacent section. If all the operations of train T_{k+1} are inserted, thus u^k is updated to u^{k+1} and iteration $k+1$ of FSP is finished.

Furthermore, with satisfying the constraints on blocking conditions and the limited capacity of resources, the time-determination procedure for one operation (the movement/traversal of a train across a section) in the above *Steps 3* and *4* is the core of the FSP algorithm. The pseudocode of time-determination procedure with key formulae will be presented in Section 6.4.3. The implementation and validation of the proposed FSP algorithm will be discussed in the next chapter.

6.4 No-Wait Blocking Parallel-Machine Job-Shop Scheduling

In practice, some prioritised trains such as passenger trains should traverse continuously without any interruption from the departure to the terminal. In this case, no-wait constraints should be satisfied, implying that these prioritised trains should enter the next section immediately after traversing on the previous section is completed throughout the whole journey.

For convenience, in this study, we use *passenger trains* to denote the *prioritised* trains that hold *no-wait* constraints, while *freight trains* is employed to signify the *non-prioritised* trains that hold *blocking* constraints.

Freight (non-prioritised) trains may be thought of as a relaxation of passenger (prioritised) trains as in comparison they may be allowed to remain in a section until the next section on the routing becomes available. In a sense, freight trains can be treated as *blocking* jobs while passenger trains are defined as *no-wait* jobs. This class of train scheduling problem, in which freight trains and passenger trains are considered simultaneously, can be regarded as a *No-Wait Blocking Parallel-Machine Job-Shop Scheduling* (NWBPMJSS) problem. The NWBPMJSS problem is a generalised case of the BPMJSS problem discussed above and is much more complicated due to the fact that no-wait constraints are more restrictive than blocking constraints.

6.4.1 Mathematical Formulation for NWBPMJSS

Based on the mathematical programming for BPMJSS in Section 6.3.1, the mathematical formulation for NWBPMJSS is completed by adding the *no-wait* constraints defined by Eq. (6.24).

NWBPMJSS Model

$$Minimise \quad C_{\max}$$

The objective function is to minimise the makespan.

Subject to:

……. (6.2-6.8)

……. (6.23)

$$\sum_{l=1}^{h_k}\sum_{k=1}^{m} r_{iolk} D_{ilk} = \sum_{l=1}^{h_k}\sum_{k=1}^{m} r_{i,o+1,l,k} s_{ilk} \ , \ o = 1,2,\ldots,m-1, \forall i \in J_{\mathrm{P}} \quad (6.24)$$

In addition to constraints defined by Eqs. (6.2-6.8) and (6.23) for the BPMJSS model in Section 6.3.1, Eq. (24) defines the no-wait constraints particularly for the NWBPMJSS model, in which J_{P} is denoted as the subset of passenger trains (or prioritised jobs). To satisfy the no-wait constraints, for each operation that belongs to a prioritised train, the departure time of this operation should be equal to the starting time of its same-prioritised-train successor.

6.4.2 Feasibility Analysis for NWBPMJSS

The *no-wait* constraints restrict that the processing of each job (train) should be continuous from the start to the end without interruptions. In other words, once a train begins to run, there must not be any idle time between any two consecutive operations of this train. Equivalently, the difference between the completion time of the last operation of a train and the starting time of its first operation is equal to the total traversing times of this train. This implies, when necessary, that the start of a train on the first traversing section may be postponed in order that each operation's completion coincides with the start of the next operation on the subsequent section.

Therefore, to satisfy the no-wait constraints, it is critical to tune up the starting time of the first operation for each train. Embedded into the feasibility-satisfaction procedure (FSP) algorithm, a tune-up algorithm is developed to guarantee that the start of a train on a certain section has to be postponed iteratively until corresponding operation's completion time equals the start of the next operation on the subsequent section.

Assuming that operation $O_{PJ[k]}$ is processed on upstream section S_j and operation O_k is processed on downstream section S_{j+1}, one situation in the tune-up procedure is illustrated in Figure 6-10, in which the starting time of $O_{PJ[k]}$ is tuned up in order to satisfy the no-wait conditions.

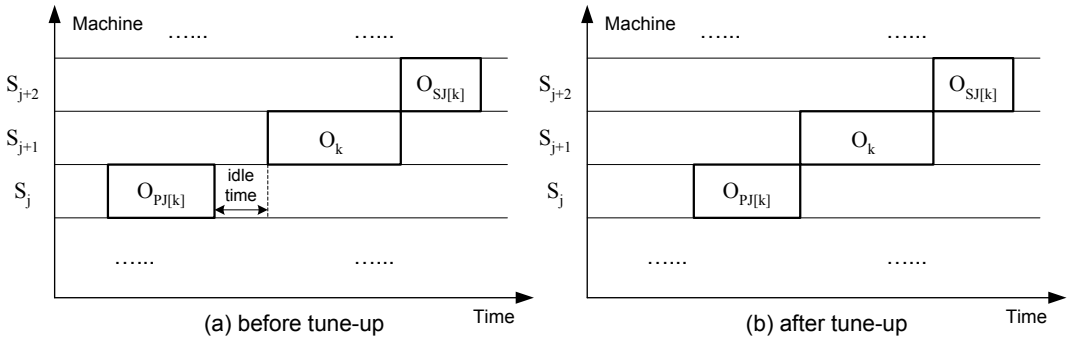

Figure 6-10: Comparison between (a) before tune-up and (b) after tune-up

However, after tunning up the starting times of operations in job-shop environments (e.g. when considering inbound and outbound trains), it is nearly inevitable to cause such conflicts that two jobs (trains) are processed simultaneously on the same machine (section) at a time. For illustration, a numerical example is given as below for analysing the conflicting situations caused after the tune-up procedure for satisfying the no-wait constraints. Firstly, shown in Figure 6-11, a feasible schedule is obtained for a 3-train 19-section NWBPMJSS case.

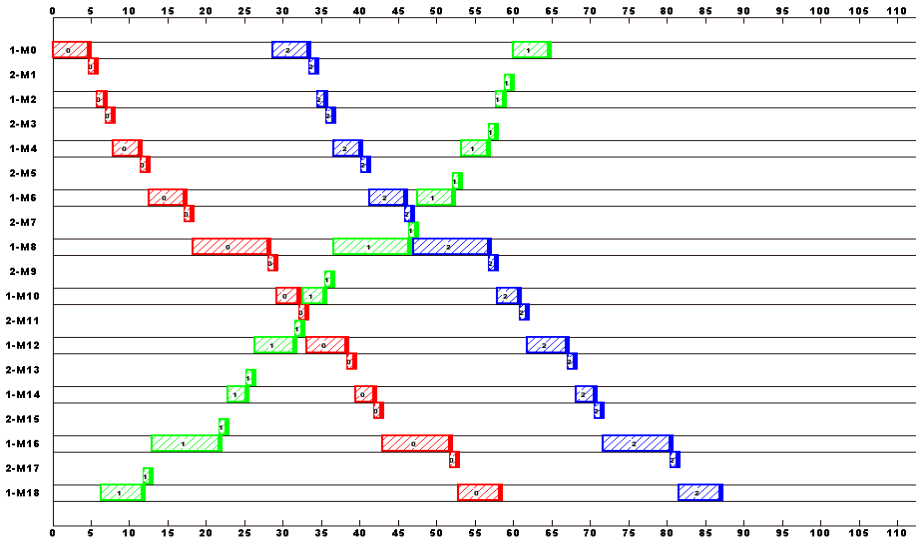

Figure 6-11: The feasible schedule for a 3-train 19-section NWBPMJSS case.

If the fourth train (job) is added, the new schedule obtained by the tune-up algorithm becomes infeasible, because a conflict happens on section (machine) 1-M16 in Figure 6-12. From the computational results, the starting time of train 0 on section 1-M16 is **42.88** while the leave time of train 3 on section 1-M16 is **43.16**, as highlighted in Figure 6-12. This implies that train 3 and train 0 will run simultaneously in the same track at a time.

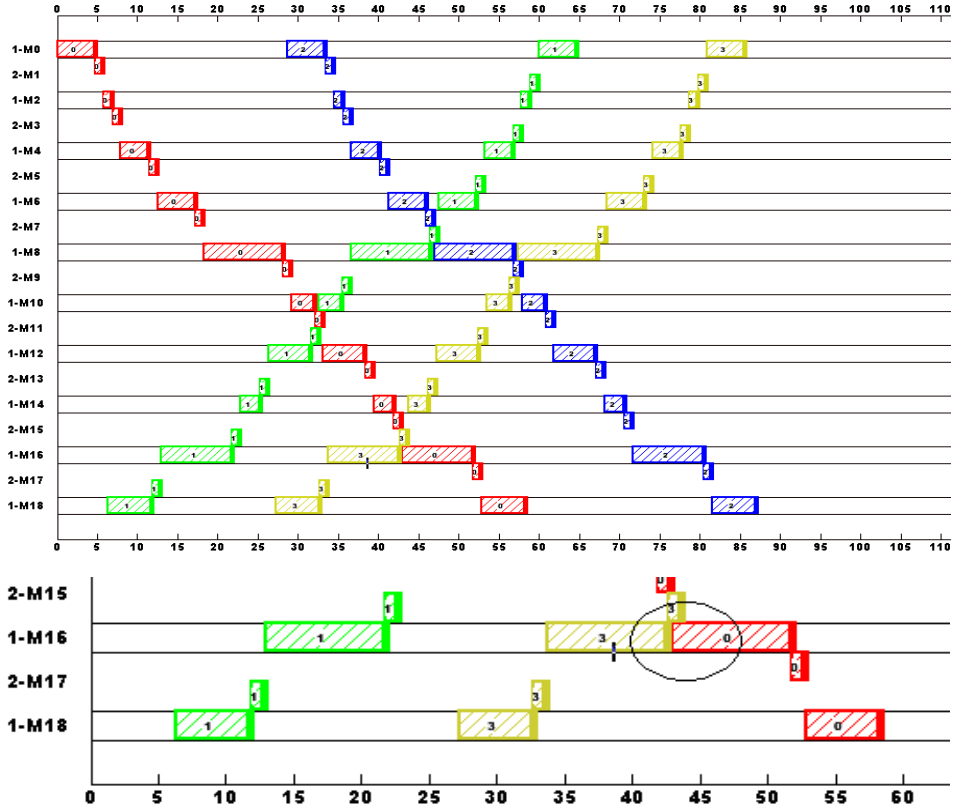

Figure 6-12: The infeasible schedule for a 4-train 19-section NWPMJSS case; note that a conflict occurs on section 1-M16.

In train scheduling environments, this situation represents a dangerous clash accident that should be prevented.

6.4.3 SE Algorithm for NWBPMJSS

With exploring the structural properties of the NWBPMJSS problem to simultaneously satisfy the blocking and no-wait constraints, an innovative algorithm called the ***SE algorithm*** is proposed to construct the feasible solution of the NWBPMJSS problem.

The basic procedure of the SE algorithm is briefly described in the following.

Step 1: Apply Module I (See Section 6.4.3.1) to schedule the first train; and then obtain the current information about train timetables and section status.

Step 2: From the second to the last in the given sequence of trains, do

2.1: Get the necessary information of the new train to be added, such as *train index, train direction* (e.g. outbound or inbound) and *train priority* (e.g. passenger train or freight train).

2.2: Switch according to the below train category:

Case 1: If the train is an outbound-freight train, apply Module II (See Section 6.4.3.2).

Case 2: If the train is an inbound-freight train, apply Module III (See Section 6.4.3.3).

Case 3: If the train is an outbound-passenger train, apply Module IV (See Section 6.4.3.4)

Case 4: If the train is an inbound-passenger train, apply Module V (See Section 6.4.3.5)

2.3: After determining the timetable of the current train, apply Module VI (See Section 6.4.3.8) to update the information of section status.

6.4.3.1 Module I of SE Algorithm

The procedure of Module I is described as below.

Step 1: Input the necessary data for scheduling the first train.

Step 2: If the direction of the first train is outbound, do the following sequentially:

2.1: Initialise the departure time for the same-job (or same-train) predecessor for the given current operation as zero, i.e. $D_{PJ} = 0$.

2.2: For $k \leftarrow 1$ (the first section) to $k \leftarrow m$ (the last section), do

2.2.1: Set the index of current operation: O_i.

2.2.2: Set the index of the section (machine) unit specially used for outbound trains, l.

2.2.3: Set the starting time e_i, completion time C_i, blocking time B_i, departure time D_i, and leave time L_i of operation O_i sequentially by Formulae (Set 1).

$$\begin{aligned} e_i &= \max(0, D_{PJ}) \\ C_i &= e_i + p_i \\ B_i &= 0 \\ D_i &= C_i \\ L_i &= D_i + \Omega_i \end{aligned} \qquad \text{(Set 1)}$$

where p_i and Ω_i are the given processing time and occupying time of operation O_i, respectively. Note that occupying time is the additional running time caused by the train length.

2.2.4: Reset $D_{PJ} = D_i$.

Step 3: Otherwise if the direction of the first train is inbound, basically apply the same method in *Step 2* but in a reverse order (i.e. from $k = m$ (the last section) to 1 (the first section)). In addition, the multiple-section (parallel-machine) units to be considered are specially assigned for inbound trains.

6.4.3.2 Module II of SE Algorithm

If the train to be added is an outbound-freight (OF) train, apply Module II to obtain the new feasible train timetables by satisfying the blocking constraints.

The procedure of Module II is described as below:

Step 1: Input the data required for scheduling this new outbound-freight train.

Step 2: Initialise the completion time for the same-job (or same-train) predecessor of the current operation as zero, i.e. $C_{PJ} = 0$.

Step 3: For $k \leftarrow 1$ (the first section) to $k \leftarrow m$ (the last section), do

3.1: Set the index of current operation: O_i ; and the index of its predecessor if it exists: $O_{PJ[i]}$.

3.2: Determine the starting time of operation O_i accordingly:

3.2.1: If the current section is a multiple-track section, apply Formulae (Set 2).

$$\begin{aligned} A_{kl^*} &= \min_{l=1,2,..,u_k^{outbound}} A_{kl} \\ e_i &= \max(C_{PJ}, A_{kl^*}) \end{aligned} \quad \text{(Set 2)}$$

where A_{kl} is the available time of the l^{th} unit on section k, l^* is the index of the chosen unit that leads to the earliest available time, A_{kl^*} is the earliest available time, and $u_k^{outbound}$ is the number of multiple-section units assigned for outbound trains.

3.2.2: Otherwise if the current section is a single-track section, apply *OFTrain-Insertion algorithm* (See section 6.4.3.3).

3.3: Set the completion time of operation $O_i : C_i = e_i + p_i$.

3.4: Set the blocking time $B_{PJ[i]}$, departure time $D_{PJ[i]}$ and leave time $L_{PJ[i]}$ of operation $O_{PJ[i]}$, by Formulae (Set 3).

$$\begin{aligned} B_{PJ[i]} &= \max(0, e_i - C_{PJ}) \\ D_{PJ[i]} &= C_{PJ[i]} + B_{PJ[i]} \\ L_{PJ[i]} &= D_{PJ[i]} + \Omega_{PJ[i]} \end{aligned} \quad \text{(Set 3)}$$

3.5: If $k = m$, apply Formulae (Set 4) to set the blocking time, departure time and leave time of the last operation O_{Last} under the assumption that the capacity of buffer storage behind the last section is unlimited.

$$\begin{aligned} B_{Last} &= 0 \\ D_{Last} &= C_{Last} \\ L_{Last} &= D_{Last} + \Omega_{Last} \end{aligned} \quad \text{(Set 4)}$$

3.6: Reset $C_{PJ} = C_i$.

6.4.3.3 OFTrain-Insertion Algorithm

In *Step 3.2.2* of Module II, the so-called *OFTrain-Insertion algorithm* is applied to feasibly determine the best starting time of a given operation O_i that belongs to a outbound-freight train traversed on a single-track section k.

The description of *OFTrain-Insertion algorithm* is given as below.

Step 1: Set the ready time of operation O_i by Formulae (Set 5).

$$\begin{aligned} A_{k+1,l^*} &= \begin{cases} \min\limits_{l=1,2,..,u_{k+1}^{outbound}} A_{kl} & if\ k < m \\ 0 & if\ k = m \end{cases} \\ R_i &= \max(C_{PJ[i]}, A_{k+1,l^*} - p_i) \end{aligned} \qquad \text{(Set 5)}$$

where A_{k+1,l^*} is the earliest available time of the downstream section of section k; R_i is defined as the feasible ready time of operation O_i; and $C_{PJ[i]}$ is the completion time of the predecessor.

Step 2: Determine the best feasible starting time of operation O_i, by exclusively considering the following three cases in sequence.

Case 1: Consider to insert operation O_i before the first scheduled operation on section k, by Formulae (Set 6).

$$\begin{aligned} I_F &= e_{[1]} - R_i \\ e_i &= R_i \ if\ p_i + \Omega_i \le I_F \end{aligned} \qquad \text{(Set 6)}$$

where I_F is the interval time between R_i (the ready time of operation O_i) and $e_{[1]}$ (the starting time of the *first* scheduled operation).

Case 2: Consider inserting operation O_i as early as possible in the middle of each pair of two scheduled operations, by Formulae (Set 7).

$$\begin{aligned} I_M &= \min(e_{[s+1]} - L_{[s]}, e_{[s+1]} - R_i) \\ e_i &= \max(R_i,\ L_{[s]})\ if\ p_i + \Omega_i \le I_M \\ &\forall s = 1, 2, \ldots n_k - 1 \end{aligned} \qquad \text{(Set 7)}$$

where I_M is the feasible interval time between each pair of two scheduled operations; $e_{[s]}$ and $L_{[s]}$ are the starting time and leave time of the s^{th} scheduled operation, respectively; and n_k is currently the number of scheduled operations on this single-track section k

Case 3: Finally, consider inserting operation O_i behind the last scheduled operation: $e_i = \max(R_i, L_{[n_k]})$ where $L_{[n_k]}$ is the leave time of the last scheduled operation.

6.4.3.4 Module III of SE Algorithm

If the train to be added is an inbound-freight train, apply Module III to obtain the new feasible train timetables. As the train direction is inbound but is non-prioritised, it basically uses the same procedure and formulae as given in Module II but the processing order of operations is reverse, i.e. for $k \leftarrow m$ (the last section) to $k \leftarrow 1$ (the first section). In addition, the multiple-section (parallel-machine) units to be considered are predestinated for inbound trains.

6.4.3.5 Module IV of SE Algorithm

If the train to be added is an outbound-passenger train, apply Module IV to obtain the new feasible train timetables by satisfying the *no-wait* constraints. Compared with Module II, Module IV is much more sophisticated due to the fact that the no-wait constraints are more restrictive than the blocking constraints. In this case, the *tune-up algorithm* (See section 6.4.3.6) is applied to satisfy the no-wait constraints. However, conflicting situations in which two operations are being processed on the same section at the same time may occur. Embedded in tune-up procedure, the *conflict-resolve algorithm* (See section 6.4.3.7) is recursively implemented for checking and resolving the conflicts.

The procedure of Module IV is described in the following:

Step 1:Input the data required for scheduling this new outbound-passenger train.

Step 2: Initialise the completion time for the same-job (or same-train) predecessor of the given current operation as zero, i.e. $C_{PJ} = 0$.

Step 3: Initialise the index of the latest fixed section: $\hat{m} = 1$, while assuming that the number of total sections is m.

Step 4: While $\hat{m} <= m$, do the following iteration:

4.1: Set the index of the current section to be considered: $k = \hat{m}$.

4.2: Set the index of the current operation processed on section k: O_i.

4.3: Determine the starting time of the current operation: e_i.

4.3.1: If the current section is a multiple-track section, apply Formulae (Set 8).

$$\begin{aligned} A_{kl^*} &= \min_{l=1,2,..,u_k^{outbound}} A_{kl} \\ e_i &= \max(C_{PJ}, A_{kl^*}) \end{aligned} \qquad \text{(Set 8)}$$

where A_{kl} is the available time of the l^{th} unit on section k, l^* is the index of the chosen unit that leads to the earliest available time, and C_{PJ} is the completion time of the predecessor.

4.3.2: Otherwise if the current section is a single-track section, apply *OFTrain-Insertion algorithm* (See section 6.4.3.3).

4.4: Set the completion time, blocking time, departure time, leave time of operation O_i sequentially by Formulae (Set 9).

$$\begin{aligned} C_i &= e_i + p_i \\ B_i &= 0 \\ D_i &= C_i \\ L_i &= D_i + \Omega_i \end{aligned} \qquad \text{(Set 9)}$$

4.5: If $e_i > C_{PJ}$ implying that no-wait conditions are not satisfied, it is necessary to apply the *tune-up algorithm* (See section 6.4.3.6) to adjust the starting times of operations scheduled at this stage. In the tune-up procedure, as the conflict may occur on any machine, apply the *conflict-resolve algorithm* (See section 6.4.3.7) to check and resolve the potential conflicts. Note that the new latest fixed section $\hat{m}$ should be recursively changed if conflicts occur in tune-up procedure.

4.6: Update C_{PJ} that equals the completion time of the operation processed on the new latest fixed section.

4.7: Consider the downstream section of the new latest fixed section in terms of train direction.

6.4.3.6 Tune-Up Algorithm

In *Step 4.3.2* of Module IV, the so-called *tune-up algorithm* is developed to recursively satisfy the no-wait constraints for a passenger train.

Assume that the current operation O_i associated with an outbound-passenger train traversing on section k ($\forall k > 1$) has just been fixed. If the starting time of the current operation is larger than the completion time of its same-job predecessor, the tune-up algorithm is applied to satisfy the no-wait constraints for the scheduled operations at the current stage.

The procedure of *tune-up algorithm* is described as follows.

Step 1: Input the datum required for applying the tune-up algorithm, such as the index of current operation: O_i and the index of the current section: k.

Step 2: For outbound trains, from $h = k$ to $h = 2$, do:

2.1: Get the starting time of the current operation processed on section h: e_h.

2.2: Get the processing time and occupying time of the same-job predecessor of operation:: $p_{PJ[h]}$ and $\Omega_{PJ[h]}$.

2.3: Tune up the timetables of the predecessor iteratively in such that there is no idle time between each pair of scheduled jobs, by Formulae (Set 10).

$$\begin{aligned} e'_{PJ[h]} &= \max(e_h - p_{PJ[h]}, e_{PJ[h]}) \\ C'_{PJ[h]} &= e_h \\ D'_{PJ[h]} &= C_{PJ[h]} \\ L'_{PJ[h]} &= D_{PJ[h]} + \Omega_{PJ[h]} \end{aligned} \qquad \text{(Set 10)}$$

2.4: After tuning up the predecessor, the conflict may occur on a single-track section h. Hence, the *conflict-resolve algorithm* is applied to check and resolve conflicts and to guarantee that only one train can be traversed in a section at a time.

2.5: After resolving the potential conflict, recursively consider the next machine by resetting $h = h - 1$ (for outbound trains).

6.4.3.7 Conflict-Resolve Algorithm

The *conflict-resolve algorithm* is proposed to resolve the conflicts that potentially occur after tuning up one operation. In a conflict, two operations are processed on the same machine (section) at a time, implying that a clash accident may occur in train scheduling environments. Obviously, this conflict should be prevented in real-life train scheduling.

The procedure of *conflict-resolve algorithm* is described as follows.

Step 1: Check whether there is a conflict on the given section h.

1.1: Get the index of operation that is just turned up on this section: O_h.

1.2: Initialise the Boolean value as false: $bIsConflic = false$.

1.3: Get the number of operations that have been scheduled on this section before tuning up operation O_h: n_h.

1.4: For $j \leftarrow n_h$ to $j \leftarrow 1$, do

1.4.1: If the below condition in Formulae (Set 11) is met, it is found that there is a conflict on section h.

$$\begin{aligned} &if\ ((e_{[j]} \le e'_h \,\&\&\, e'_h \le L_{[j]}) \,||\, (e_{[j]} \le L'_h \,\&\&\, L'_h \le L_{[j]})) \\ &\quad bIsConflic = true \\ &\quad j^* = j \end{aligned} \qquad \text{(Set 11)}$$

where $e_{[j]}$ and $L_{[j]}$ are the starting and leaving time of the j^{th} scheduled operation on the given section; e'_h and L'_h are the new starting and leaving time of the tuned-up operation O_h; $\&\&$ and $||$ are operators (*conditional AND*) and (*conditional OR*), respectively; j^* is the position index of the chosen scheduled operation that is conflicting with the turned-up operation.

1.4.2: If $bIsConflic = false$, continue to consider the next scheduled operation by resetting $j = j - 1$; otherwise, stop checking and go to *Step 2*.

Step 2: If $bIsConflic = true$, resolve the conflict.

2.1: Reset the starting time of the tuned-up operation: e'_h .

For $\tau \leftarrow j^*$ to $\tau \leftarrow n_h - 1$, apply Formulae (Set 12).

$$\begin{aligned} & e'_h = L_{[\tau]} \quad if \ ((e_{[\tau+1]} - L_{[\tau]}) > (p_h + \Omega_h)) \\ & \tau = \tau + 1 \quad otherwise \end{aligned} \tag{Set 12}$$

where $L_{[\tau]}$ is the leave time of a scheduled operation and $e_{[\tau+1]}$ is the starting time of its successor; p_h and Ω_h is the processing time and occupying time of the tuned-up operation O_h .

If the above condition in (Set 12) is not met and τ reaches n_r , then set $e'_h = L_{[n_h]}$.

2.2: Reset the completion time, departure time and leave time of the turned-up operation by Formulae (Set 13).

$$\begin{aligned} & C'_h = e'_h + p_h \\ & D'_h = C'_h \\ & L'_h = D'_h + \Omega_h \end{aligned} \tag{Set 13}$$

2.3: After resolving the conflict on section h , it is crucial to update the index of the latest fixed machine to be used in Module (IV), by setting $\hat{m} = h$. Then, go to the next iteration at *Step 4* of Module IV that will start from the new machine $\hat{m}$ on which the conflict is newly resolved.

Step 3: If $bIsConflic = false$, update $\hat{m} = \hat{m} + 1$ and go to the next iteration at *Step 4* of Module IV.

6.4.3.7 Module V of SE Algorithm

If the train to be added is an inbound-passenger train, apply Module V to obtain the new feasible train timetables. In likeness, the procedure and formulae employed in

Module V is similar to those in Module IV but the processing order of operations is in the opposite direction. In addition, the multiple-section (parallel-machine) units to be considered are appointed for inbound trains.

6.4.3.8 Module VI of SE Algorithm

After determining the new train timetables by adding a new train, Module VI is applied to update the information of section status.

The procedure of Module (VI) is briefly described as follows.

Step 1: For $k \leftarrow 1$ to $k \leftarrow m$, do:

2.1: Get the index of operation processed on section k and associated with the added train: O_i.

2.2: Increase the number of scheduled operations on section k, by setting $n_k = n_k + 1$.

2.3: If section k is a multiple-track section,

2.3.1: Update the chosen unit of section k : l_k^*

2.3.2: Update the available time of this unit: $A_{kl^*} = L_i$

2.4: If section k is a single-track section,

2.4.1: Rearrange the sequence of operations scheduled on this single-track section k.

2.4.2: Update the starting time of operation O_i.

2.4.3: Update the leave time of operation O_i.

In terms of various settings of train directions (outbound or inbound) and train priorities (freight or passenger), a class of train scheduling problems can be classified into the below six cases.

(1) *Blocking Parallel-Machine Flow-Shop-Scheduling* (BPMFSS)
(2) *Blocking Parallel-Machine Job-Shop-Scheduling* (BPMJSS)
(3) *No-Wait Parallel-Machine Flow-Shop-Scheduling* (NWPMFSS)
(4) *No-Wait Blocking Parallel-Machine Flow-Shop-Scheduling* (NWBPMFSS)

(5) *No-Wait Parallel-Machine Job-Shop-Scheduling (*NWPMJSS)

(6) *No-Wait Blocking Parallel-Machine Job-Shop-Scheduling* (NWBPMJSS)

Because NWBPMJSS is the most generalised case, the proposed *SE* algorithm is very generic because it is able to solve all of these six train scheduling problems by only adjusting the attributes in data input. This superior merit of the *SE* algorithm will be validated in computational experiments in Appendix 8.

6.4.4 SE-BIH Algorithm for NWBPMJSS

Similar to the structure of the *LK-BIH* algorithm (see Section 5.7) by combining the constructive algorithm (i.e. the *SE* algorithm) and the local-search heuristic (i.e. the *Best-Insertion-Heuristic* algorithm), a two-stage hybrid heuristic algorithm called the *SE-BIH algorithm* is developed to construct a good NWBPMJSS solution. The procedure of the *SE-BIH* algorithm for solving the NWBPMJSS problem is described as follows.

Step 1: Build up a partial solution M with only one initial job that has the longest processing time.

Step 2: For $k \leftarrow 2$ to $k \leftarrow n$, do

2.1: At step k, consider all possible combinations of $n-k+1$ jobs with k insertion positions to obtain a set of alternative permutation sequences of k jobs.

2.2: Apply SE algorithm to these alternatives and obtain the feasible NWBPMJSS solutions.

2.3: Select M'^{*} that is the best NWBPMJSS solution with the minimum makespan.

2.4: Update the partial solution, i.e. $M = M'^{*}$. If M is complete, stop; otherwise, go to Step 2 for the next iteration.

In a sense, the *SE-BIH* algorithm considers all possible insertions of all not-yet-included jobs while successively building a feasible NWBPMJSS schedule. That is, starting with an initial job that has the longest processing time, at each step

$k = 2,...,n$, all possible combinations of $n-k+1$ jobs with k insertion positions are investigated by the *SE* and *BIH* algorithms.

6.4.5 Tabu Search for NWBPMJSS

The *SE* algorithm is proposed to construct a feasible NWBPMJSS schedule in terms of the given permutation of trains. To minimise the makespan, it is critical to find the best sequence of trains. This can be achieved by neighbourhood search guided by *Tabu Search* (TS), which is a metaheuristic algorithm designed for finding the optimal (or near-optimal) solution for many combinatorial optimisation problems.

6.4.5.1 Description of Tabu Search

TS algorithm consists of several elements called the *initial solution*, *move*, *neighbourhood*, *searching strategy*, *memory*, *aspiration function*, *stopping rules*, etc. The *move* is a function which transforms a solution into another solution. The subset of moves applicable to a given solution generates a collection of solutions called the *neighbourhood*. TS starts from an initial feasible solution constructed by the *SE* algorithm. At each step, the neighbourhood of the current solution is searched in order to find an appropriate neighbour, typically the best in the neighbourhood. Next, the move that leads to this neighbour is performed and the resulting solution becomes the new current solution to initiate the next step. To avoid cycling or implicitly becoming trapped in a local optimum, a short term memory called a *tabu list* determines moves that are forbidden. Attributes of moves are identified to be recorded on this list, for a chosen span of time, as a way to prevent future moves that would "undo" the effects of previous moves. Nevertheless, a forbidden move may be accepted if an aspiration function evaluates it as sufficiently profitable. The stopping rule that ends the search traditionally consists of setting a limit on the execution time, number of iterations, and number of consecutive iterations without improving the makespan.

The brief description of the basic TS procedure is given below.

Step 1: Generate an initial solution s

Step 2: Initialise the tabu list.

Step 3: Perform a certain number of TS iterations:

3.1: Build up the neighbourhood based on the current solution s and solve the neighbours (e.g. the NWBPMJSS schedules) feasibly by the corresponding constructive algorithm (e.g. the *SE* Algorithm).

3.2 Choose a best neighbour s^*, which is not a tabu or satisfies the aspiration criterion.

3.3 Set $s = s^*$ and update the tabu list. If the stopping condition is met, go to *Step 4*.

Step 4: Return the best solution found.

Since TS is a general approach, its application needs detailed definitions of basic elements, among which the design of neighbourhood is the most important part, because a good neighbourhood structure can guide the search to find the better solutions in a new space.

The analysis and definition of neighbourhood structure for searching the better NWBPMJSS solution are given in the next section.

6.4.5.2 The Definition of Neighbourhood Structure

In NWBPMJSS, the solution is represented by a permutation sequence of trains. The neighbourhood of a permutation can be created by devising a set of permutation moves. Among many types of permutation moves considered in the literature, two of them appear prominently:

(1) exchange jobs placed at the a^{th} and b^{th} positions (called an *E-Move*);

(2) remove a job placed at the a^{th} position and insert it at the b^{th} position (called an *I-Move*).

Taillard (1990) experimentally showed that E-Moves are not better than I-Moves to find good permutations. Therefore, we will consider only I-Moves, illustrated as follows.

Let $v=(a,b)$ be a pair of positions $a,b\in\{1,\ldots,n\}$, $a\neq b$ in the given permutation $\Pi=(\pi(1),\ldots\pi(a-1),\pi(a),\pi(a+1),\ldots,\pi(b),\pi(b+1),\ldots,\pi(n))$.

Such a permutation move is defined as I-Move by a pair of position $v=(a,b)$.

- If $a<b$, the new permutation sequence is:

  $$\Pi_v=(\pi(1),\ldots\pi(a-1),\pi(a+1),\ldots,\pi(b),\pi(a),\pi(b+1),\ldots,\pi(n))$$

- If $a>b$, the new permutation sequence is:

  $$\Pi_v=(\pi(1),\ldots\pi(b-1),\pi(a),\pi(b),\ldots,\pi(a-1),\pi(a+1),\ldots,\pi(n))$$

Each pair $v=(a,b)$ defines a move from the current permutation Π and let V be a set of such pairs. The neighbourhood is denoted as $N(V,\Pi)=\bigcup\Pi_v \mid v\in V$, where a set of moves is defined as $V=\{(a,b):b\in\{a-1,a\},a,b\in\{1,\ldots,n\}\}$.

It is observed that two moves (i.e. $v_1=(a,b)$ and $v_2=(b,a)$) yield the same permutation if $|a-b|=1$. In order to avoid redundancy, V contains exactly one from each pair of equivalent moves.

Thus, the neighbourhood $N(V,\Pi)$ has $(n-1)^2$ neighbours and satisfied a *connectivity property*: for any $\Pi^{(1)}$ there exists a finite sequence $\Pi^{(1)},\Pi^{(2)},\ldots,\Pi^{(r)}$ such that $\Pi^{(r)}$ is an optimal solution and $\Pi^{(i+1)}\in N(V,\Pi^{(i)})$, where $\Pi^{(i)}$ is the current permutation sequence at the i^{th} TS iteration.

The large-size neighbourhood requires considerable computational effort for the search. In the literature, a common tendency is to evaluate neighbours by lower bounds instead of calculating the makespan exactly. In NWBPMJSS, however, neighbours have to be explicitly computed by the *SE* algorithm due to feasibility requirements and hence it is preferred to use smaller-size neighbourhood moves that lead to better solutions. A promising approach to the problem of reducing computational effort is proposed by exploiting the *block properties* on the *critical* sequence.

A critical sequence is the permutation sequence of trains on the so-called bottleneck section, on which it has the minimum number of gaps or idle times.

For a block $B_l = (\pi(u_{l-1}), \pi(u_{l-1}+1), \ldots, \pi(u_l))$, $\forall l = 1,2,...,k$, where it is assumed that there are k blocks in total on the critical sequence, we consider a subset of I-Moves $W_l(\Pi) = \{(a,b) \in V : a,b \in \{u_{l-1}+1, \ldots, u_l - 1\}\}$ which are performed inside the block B_l.

Let $W(\Pi) = \bigcup_{l=1}^{k} W_l(\Pi)$, we found that such an important property holds for the NWBPMJSS problem.

Property 6.1: For any NWBPMJSS sequence $\Pi' \in N(W(\Pi), \Pi)$, we have $C_{\max}(\Pi') \geq C_{\max}(\Pi)$.

Property 6.1 is proposed by Grabowski and Wodecki (2004) for the classical PFSS problem. This property implies that any new solution Π' from the neighbourhood $\Pi' \in N(W(\Pi), \Pi)$ cannot lead to the better makespan. The validity of this property for NWBPMJSS can be verified by considerable computational experiments given in Appendix 6.

According to Property 6.1, the moves from $V - W(\Pi)$ are more useful than those from $W(\Pi)$ to construct the neighbourhood structure. However, the space of neighbourhood $N(V - W(\Pi), \Pi)$ still seems to be "too big". Therefore, we need to go further in the neighbourhood reduction.

For each candidate train (job) j, we consider at most one move to the right and at most one to the left. Thus, the sets of *candidate jobs* are defined like that:

$$E_{la} = B_l - \pi(u_{l-1}) = (\pi(u_{l-1}+1), \ldots \pi(u_l - 1), \pi(u_l))$$

$$E_{lb} = B_l - \pi(u_l) = (\pi(u_{l-1}), \pi(u_{l-1}+1), \ldots, \pi(u_l - 1))$$

Each set E_{la} (or E_{lb}) contains the jobs in the l^{th} block that are candidate jobs for being moved to the left (or right) in the position immediately before job $\pi(u_{l-1})$ (or after job $\pi(u_l)$).

For any block B_l ($\forall l = 1,2,\ldots,k$) on a critical sequence, we define the following set of moves to the left: $ZL_l(\Pi) = \{(x, u_{l-1}) \mid \pi(x) \in E_{la}\}$ which contains all moves of jobs of E_{la} to the left before the first job of the l^{th} block. By analogy, $ZR_l(\Pi) = \{(x, u_l) \mid \pi(x) \in E_{lb}\}$ contains all moves of jobs of E_{lb} to the right after the last job of the l^{th} block.

Note that if $|B_l| = 2$, then $E_{la} = \{\pi(u_l)\}$ and $E_{lb} = \{\pi(u_{l-1})\}$ and $ZR_l(\Pi) = ZL_l(\Pi)$. Thus, one of them can be eliminated to reduce the redundancy. Further if $|B_l| = 1$, then $E_{la} = E_{lb} = \phi$ and $ZR_l(\Pi) = ZL_l(\Pi) = \phi$.

As a consequence of above analysis, such a concise set of moves:

$$Z(\Pi) = \bigcup_{l=1}^{k} [ZR_l(\Pi) \bigcup ZL_l(\Pi)]$$

creates the neighbourhood $N(Z(\Pi), \Pi)$ adopted for the NWBPMJSS problem.

Through a great deal of computational experiments, we found that the neighbourhood structure plays a critical role in a local search procedure. If the neighbourhood structure is powerful enough, it will be fast to find a solution close to the optimum after a finite number of iterations. With the above neighbourhood structure, the TS algorithm is developed to solve the NWBPMJSS problem efficiently. The computation experiment for the performance of the proposed neighbourhood structure $N(Z(\Pi), \Pi)$ is given in Appendix 7.

6.4.5.3 The Definition of Tabu List

In our algorithm, we use the tabu list defined as a finite list T containing ordered pairs of jobs with the dynamic length. The tabu list is defined to determine some moves that are forbidden to be chosen for a certain period.

If a move $v=(x,u_{l-1})\in ZL_l(\Pi)$ is performed on the current permutation Π to generate a neighbour Π_v, then the pair of jobs $(\pi(x-1),\pi(x))$ is added into the tabu list.

On the other hand, a move $v=(x,u_l)\in ZR_l(\Pi)$ is performed on the current permutation Π to generate a neighbour Π_v, then the pair of jobs $(\pi(x),\pi(x+1))$ is added into the tabu list.

With respect to a permutation Π, a move $v=(x,u_{l-1})\in ZL_l(\Pi)$ (or a move $v=(x,u_l)\in ZR_l(\Pi)$) has tabu status, if $TL(\pi(x))\bigcap B_l\neq\phi$ (or $TR(\pi(x))\bigcap B_l\neq\phi$), where the set of $TL(j)$ or $TR(j)$ is defined as $TL(\pi(x))=\{j\in J\mid(j,\pi(x))\in T\}$ (or $TR(\pi(x))=\{j\in J\mid(\pi(x),j)\in T\}$).

6.4.6 SE-BIH-TS Algorithms

To guarantee that the optimal (or near-optimal) solution of the NWBPMJSS problem can be obtained, a three-stage algorithm called *SE-BIH-TS* is proposed by combining the TS algorithm with the *SE-BIH* algorithm (see Section 6.6.5).

The procedure of the *SE-BIH-TS* algorithm is described as follows.

Step 1: Build up a partial solution M with only one initial job that has the longest processing time.

Step 2: For $k\leftarrow 2$ to $k\leftarrow n$, do

2.1: According to the best-insertion-heuristic (BIH) mechanism, consider all possible combinations of $n-k+1$ jobs with k insertion positions to obtain a set of alternative permutation sequences of k jobs.

2.2: Apply the *SE* algorithm to these alternatives and obtain the feasible NWBPMJSS solutions.

2.3: Select M'^{*} that is the best NWBPMJSS solution with the minimum makespan.

2.5: Try to improve the current partial schedule M'^{*} by applying the proposed TS algorithm with a few iterations, if the current neighbourhood moves are applicable.

Step 3: Apply the proposed TS algorithm to improve the complete NWBPMJSS schedule obtained by SE-BIH algorithm.

In the *SE-BIH-TS* algorithm, the TS algorithm is initialised by the best complete schedule found by the *SE-BIH* algorithm. Instead of using a random initial starting solution, this serves to find the optimal (or near-optimal) solution quickly. In addition, the partial schedule obtained by the *SE-BIH* algorithm at each step k is improved by TS with a limited number of iterations.

6.5 Summary

In general, the train scheduling problem is modelled as a *blocking parallel-machine job-shop-scheduling* (BPMJSS) problem. The BPMJSS problem is mathematically formulated and analysed in an alternative graph model. Based on the analysis, it is not trivial to construct a feasible BPMJSS schedule. Inspired by an improved *Shifting Bottleneck Procedure* for PMJSS and characteristic analysis of blocking constraints, we propose a new constructive algorithm called *Feasibility Satisfaction Procedure* (FSP) to achieve the feasible BPMJSS schedule.

Furthermore, the BPMJSS model is generalised to be a *No-Wait Blocking Parallel-Machine Job-Shop Scheduling* (NWBPMJSS) problem when considering the *prioritised* trains and *non- prioritised* trains simultaneously. In this case, no-wait conditions arise because the prioritised trains such as express passenger trains should traverse continuously without any interruption. In comparison, non-prioritised trains such as freight trains are allowed to enter the next section immediately if possible or to remain in a section until the next section on the routing becomes available, which is thought of as a relaxation of no-wait conditions.

Some properties of this methodology for train scheduling are summarised below.

1. The BPMJSS problem is the generalised case of the PMJSS problem by considering the blocking constraints due to the lack of inter-resource buffer storage.

2. The NWBPMJSS problem is the generalised case of the BPMJSS problems by adding the no-wait constraints that are more restrictive than the blocking constraints.

3. In terms of various data settings such as train directions (outbound or inbound) and train priorities (freight or passenger), the train scheduling problems are classified into the following six types:

   (1) *Blocking Parallel-Machine Flow-Shop-Scheduling* (BPMFSS)
   (2) *Blocking Parallel-Machine Job-Shop-Scheduling* (BPMJSS)
   (3) *No-Wait Parallel-Machine Flow-Shop-Scheduling* (NWPMFSS)
   (4) *No-Wait Blocking Parallel-Machine Flow-Shop-Scheduling* (NWBPMFSS)
   (5) *No-Wait Parallel-Machine Job-Shop-Scheduling (*NWPMJSS)
   (6) *No-Wait Blocking Parallel-Machine Job-Shop-Scheduling* (NWBPMJSS).

4. Due to the difficulty to satisfy the blocking and no-wait constraints, the train schedule is represented by the permutation sequence of trains at first. In terms of the given permutation sequence, the feasible timetable (i.e. time information of each operation) is constructed by the proposed constructive algorithm.

5. A generic constructive algorithm called the *SE* algorithm is proposed to construct the feasible timetable for any one of the above six train scheduling problems.

6. The proposed *SE* algorithm particularly incorporates the tune-up and conflict-resolve algorithms, in order to satisfy the feasibility of the no-wait conditions.

7. A two-stage hybrid algorithm called the *SE-BIH* algorithm is developed to construct a good complete NWBPMJSS schedule.

8. In order to further improve the solution quality of the NWBPMJSS schedule, a sophisticated neighbourhood structure is developed under the architecture of the Tabu Search metaheuristic algorithm.

9. A three-stage hybrid algorithm called the *SE-BIH-TS* algorithm is developed to find the optimal (or near-optimal) NWBPMJSS solution. In the *SE-BIH-TS* algorithm, the TS algorithm is initialised by the best complete schedule found by the *SE-BIH* algorithm.

In summary, the proposed methodology would be very promising because it can be applied as a fundamental tool for modelling and solving many real-world scheduling problems.

Chapter 7 Implementations and Validations

CHAPTER OUTLINE

7.1 Comparisons between PMJSS and BPMJSS

The following computational experiment of a real-world train scheduling case is used to compare and analyse the PMJSS and BPMJSS models. The data for this example are given as below:

Table 7-1: The sectional running times for a BPMJSS instance

Sections	Sectional Running Times									
	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
1-S0	4.75	4.75	4.75	4.75	4.75	4.75	4.75	4.75	4.75	4.75
2-S1	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02
1-S2	1.13	1.13	1.13	1.13	1.13	1.13	1.13	1.13	1.13	1.13
2-S3	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02
1-S4	3.56	3.56	3.56	3.56	3.56	3.56	3.56	3.56	3.56	3.56
2-S5	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02
1-S6	4.72	4.72	4.72	4.72	4.72	4.72	4.72	4.72	4.72	4.72
2-S7	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02
1-S8	9.93	9.93	9.93	9.93	9.93	9.93	9.93	9.93	9.93	9.93
2-S9	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02
1-S10	2.90	2.90	2.90	2.90	2.90	2.90	2.90	2.90	2.90	2.90
2-S11	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02
1-S12	5.25	5.25	5.25	5.25	5.25	5.25	5.25	5.25	5.25	5.25
2-S13	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02
1-S14	2.48	2.48	2.48	2.48	2.48	2.48	2.48	2.48	2.48	2.48
2-S15	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02
1-S16	8.93	8.93	8.93	8.93	8.93	8.93	8.93	8.93	8.93	8.93
2-S17	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02
1-S18	5.55	5.55	5.55	5.55	5.55	5.55	5.55	5.55	5.55	5.55

In this instance, six outbound trains and four inbound trains are traversing on 19 sections that consist of 10 single-track sections and 9 double-track sections. For simplicity, in this train scheduling case, the sectional running times for different trains on the same section are identical and the additional running time caused by the train's length is added into the sectional running times. At first, without considering the blocking constrains, this case is modelled as a PMJSS problem and solved by the improved SBP algorithm (see Section 6.2.3). The proposed methodology is coded by Visual C++, which is object-oriented programming that tends to produce software that is more maintainable and better organised. All typical application-specific modules are treated as the following objects: data, formulae, algorithms, output, graphics and other behaviours. Therefore, the software provides a convenient way to quickly obtain the new feasible solutions for

different type scheduling problems by only changing the attributes in the data input. In addition, the solutions can be conveniently shown and analysed in the graphic interface.

Firstly, the PMJSS result obtained by the proposed *shifting bottleneck procedure* (SBP) algorithm (Liu and Kozan, 2008b) for this test problem is displayed by the Gantt chart in Figure 7-1, in which trains are distinguished by different colours and numbers. The makespan of this PMJSS schedule is 135.78.

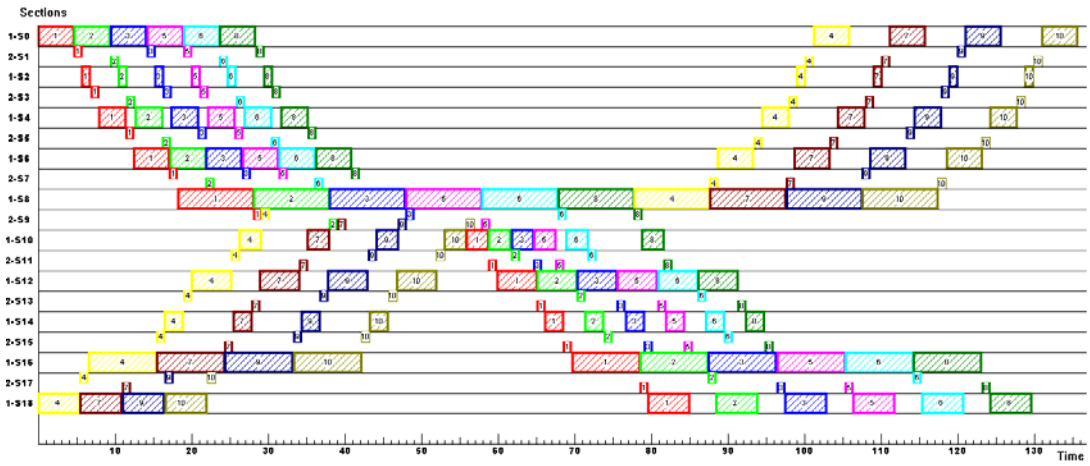

Figure 7-1: The Gantt chart for the PMJSS schedule obtained by SBP

This PMJSS result is feasible when the buffer capacity between successive machines is unlimited, i.e. without considering blocking conditions. However, it becomes infeasible if this PMJSS schedule is implemented in train scheduling environments, because in a real-life situation the hold-while-wait (blocking) constraints and deadlock situations should be considered.

For illustrating infeasibility, this PMJSS schedule is depicted in the string chart in Figure 7-2, for describing the relationship between the train's position and time point. It is easy to find from the string chart that there are many operation overlaps (i.e. train conflicts) in some sections (e.g. 2-S7, 2-S9, 2-S11, 2-S15 and 2-S17). In a real-life train scheduling case, the situation whereby more than one train will traverse on a track section at the same time is very dangerous and definitely prohibited. As a result, the PMJSS schedule shown in Figure 7-1 is not applicable to many real-life train scheduling cases.

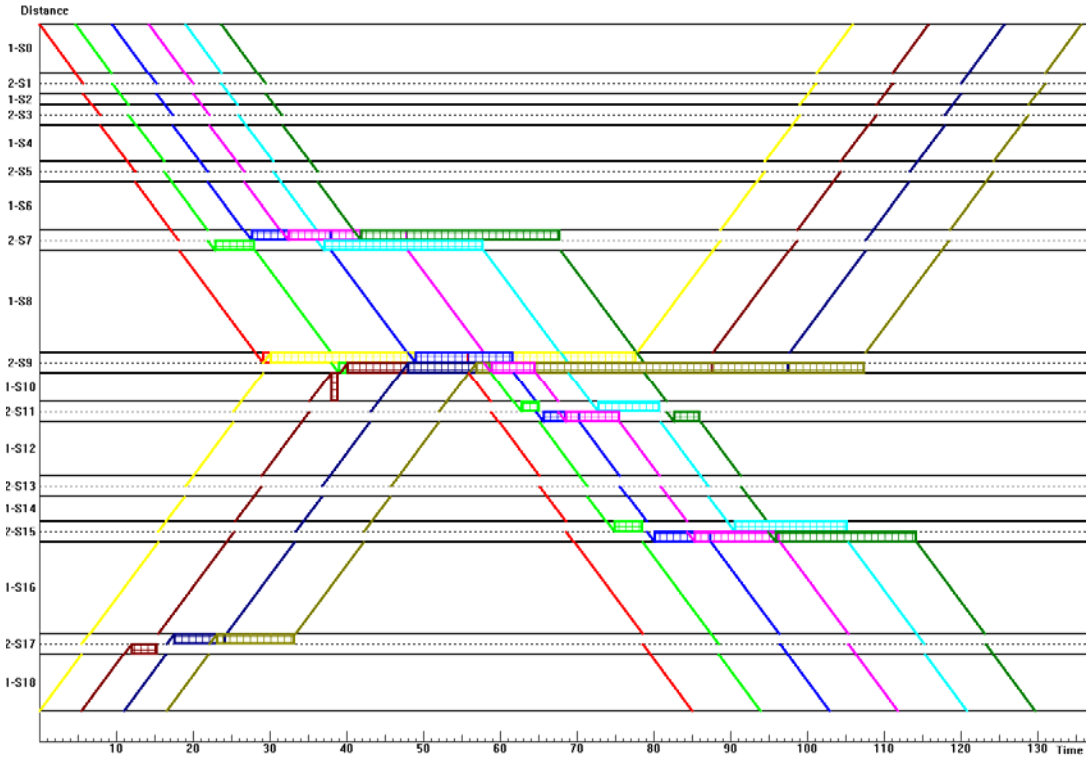

Figure 7-2: The String chart for the PMJSS schedule obtained by SBP, in which the blocking times are highlighted by cross brush

Therefore, in consideration of the blocking conditions, the train scheduling problem has to be modelled as a BPMJSS problem. The proposed BPMJSS model can be solved by a new algorithm called the *Feasibility Satisfaction Procedure* (FSP) (see Section 6.3.3). The Gantt chart of the BPMJSS schedule obtained by FSP for this test problem is drawn in Figure 7-3. The makespan of this feasible BPMJSS schedule shown in Figure 7-3 is also 135.78.

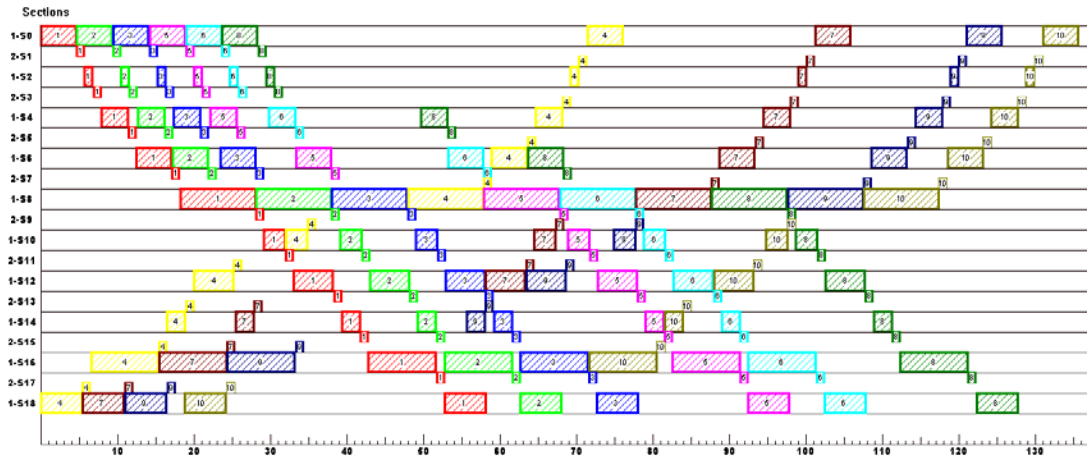

Figure 7-3: The Gantt chart for the BPMJSS schedule obtained by FSP

For the sake of analysing feasibility, the string chart for this BPMJSS schedule obtained by FSP is drawn in Figure 7-4.

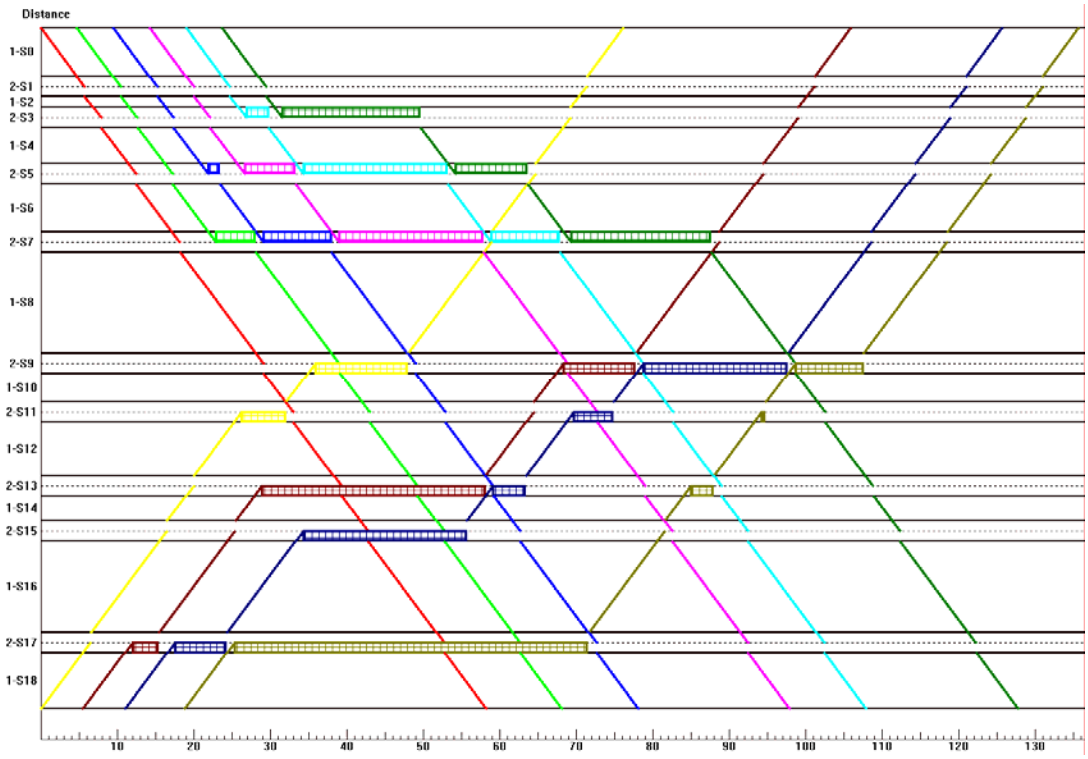

Figure 7-4: The string chart of the BPMJSS schedule obtained by FSP

In this string chart, there are no overlaps of blocking boxes highlighted by cross-brush, implying that the blocking constraints are satisfied and the obtained BPMJSS schedule is feasible or applicable for real-world implementations.

7.2 Implementations

In this section, some practical implementations of the proposed methodology described in Chapter 6 are discussed.

7.2.1 Considering Train Length

First, in practice, the train length should be considered because it has a great impact on the performance of operating a railway. This is because, when a train is

traversing from one section into the next section, the train has to occupy these two sections in a period (i.e. the occupying time caused by train length) till the whole body of the train completely leaves the section. The illustration of effects caused by the train length is given in Figures 7-5, 7-6 and 7-7.

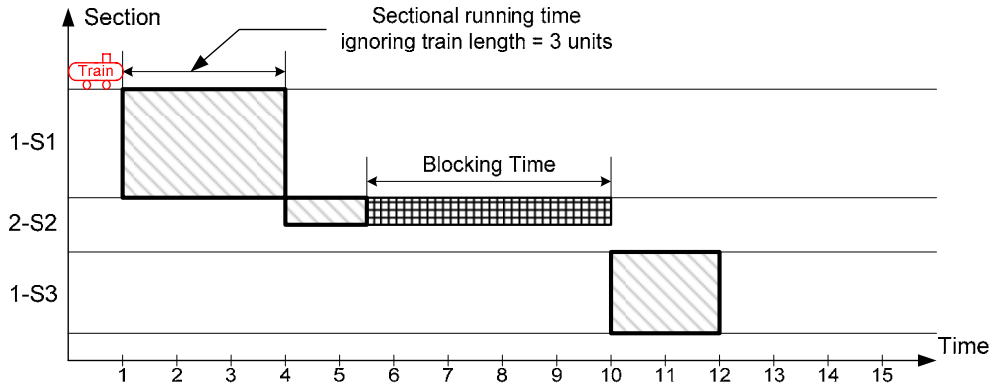

Figure 7-5: The Gantt chart when the train length is totally ignored

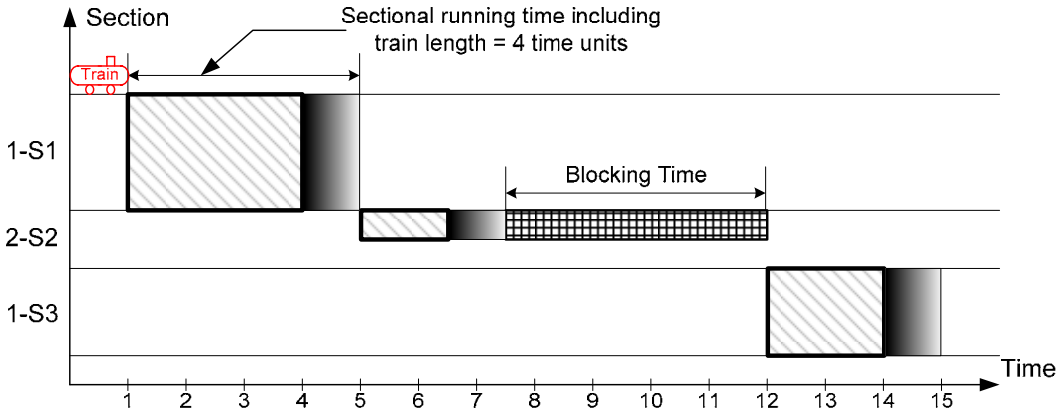

Figure 7-6: The Gantt chart when the train length is included in sectional running time

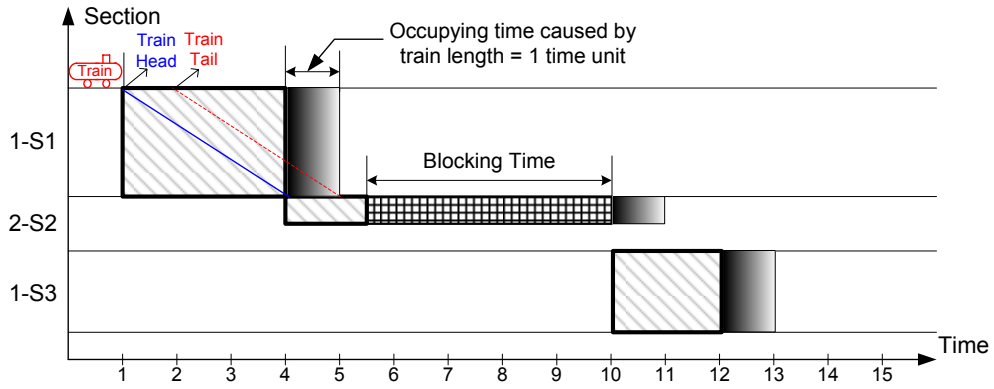

Figure 7-7: The Gantt chart when the train length is excluded from sectional running time

The first case in Figure 7-5 is not safe for real-world train scheduling when the train length is totally ignored. The second case in Figure 7-6 is applicable but it is not precise, especially when the train is very long. In comparison, the third case in Figure 7-7 is the most precise and can be used to represent real-life cases.

In addition, the makespan of the third case becomes smaller in comparison with that of the second case when the occupying time is added into the sectional running time.

The new parameters adopted in the SE algorithm to obtain the more realistic solutions with the application of the third case analysed in Figure 7-7 are listed below.

Ω_i	occupying time of operation *i*, caused by the corresponding train length
p_i'	new sectional running time of operation i ($p_i' = p_i - \Omega_i$)
c_i'	new completion time of operation i when the sectional running time excludes train length ($c_i' = c_i - \Omega_i$)
$SJ[i]$	the same-job successor of operation i
$SM[i]$	the same-machine successor of operation i
$PJ[SM[i]]$	the same-job predecessor of operation $SM[i]$
$e'_{SJ[i]}$	new starting time of operation $SJ[i]$
b_i'	new blocking period of operation i ($b_i' = e'_{SJ[i]} - c_i'$)
D_i'	new departure time ($D_i' = c_i' + b_i'$)
L_i	leave time at which the whole body of the train completely enters the next section ($L_i = D_i' + \Omega_i$)
$e'_{SM[i]}$	new starting time of $SM[i]$ ($e_{SM[i]} = \max(e_{SJ[i]}, L_{PJ[SM[i]]}, L_i)$)

To further verify the great impact of train length, the PMJSS instance given in Table 7-1 is solved by the SE algorithm with the above adjusted parameters and formulae. The corresponding BPMJSS solution is displayed in Figure 7-8, in which the boxes of occupying times are highlighted by the solid brush.

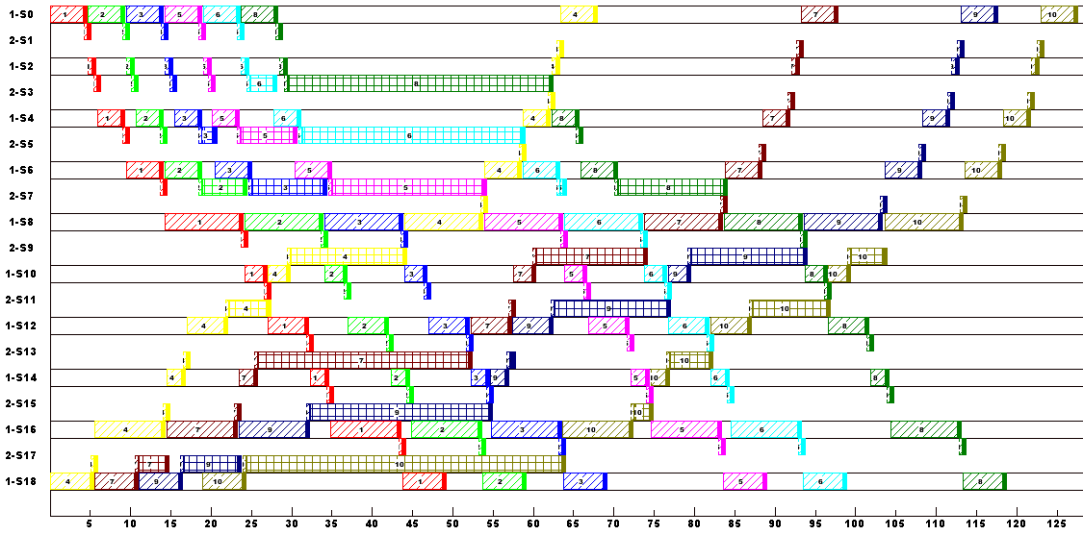

Figure 7-8: The Gantt chart for the new BPMJSS result when the sectional running time excludes train length

In this case, the occupying times (measured as 0.5 times units for all trains) caused by train length are excluded from the sectional running times for each train. It is observed that the makespan can decrease from **135.78** (see Figure 7-3) to **127.28** in Figure 7-8, although the BPMJSS schedule is still the same.

7.2.2 Upgrading Track Sections

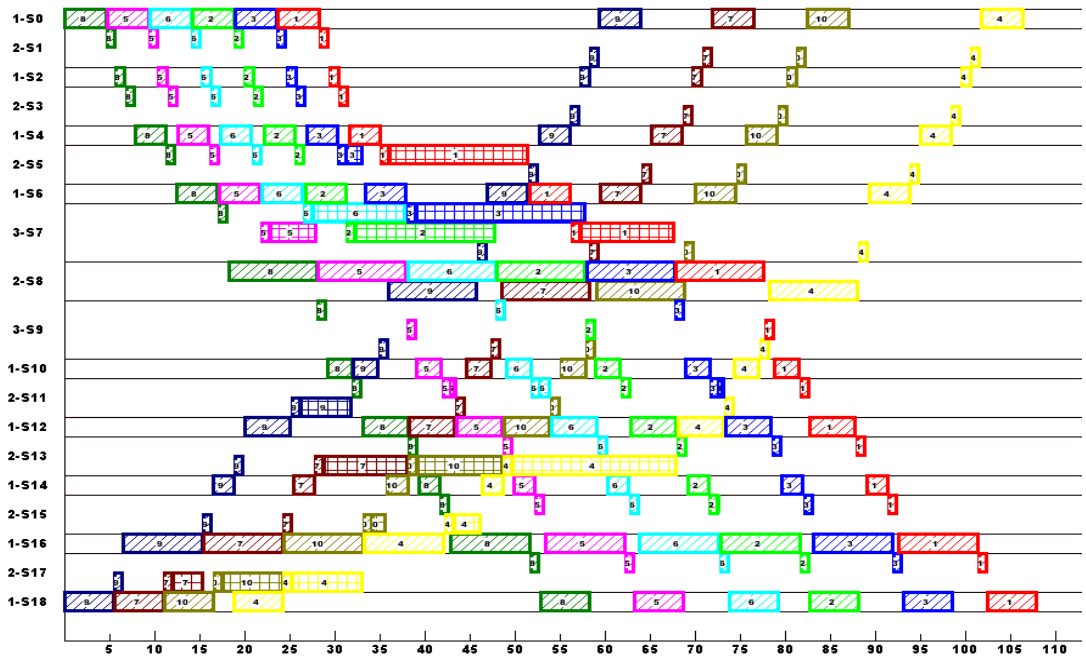

Figure 7-9: The Gantt chart of the new BPMJSS schedule when upgrading the track sections

In addition, the proposed methodology is very helpful for the decision making on upgrading the number and location of track sections. For example, a single-track section 1-S8 in Figure 7-3 is very busy because it has the minimum gaps (or idle times) between trains. Thus, it is called a ***bottleneck section***.

If this bottleneck section (1-S8) and its two adjunct double-track sections (2-S7 and 2-S9 in Figure 7-3) are upgraded by respectively adding one more track to be sections 3-S7, 2-S8 and 3-S9 in Figure 7-9, the makespan of the new BPMJSS schedule declines to **108.03** in Figure 7-9. Comparing to the original makespan value **135.78** in Figure 7-3, this is a huge improvement in the efficiency of operating the overall railing system.

7.2.3 Accelerating a Tardy Train

Moreover, the proposed methodology is suitable for dynamic scheduling. For example, assuming that Train 4 (highlighted by Yellow color) in Figure 7-9 arrives late at the destination at time point **106.51**, there is an easy way to make Train 4 arrive on time while the timetables of other trains are unchanged. This method only needs to accelerate Train 4 on Section 1-S12 in such a way that the sectional running time is smaller than the interval between Train 6 and Train 2 on this section to fit the gap. Thus, the new result shown in Figure 7-10 can be easily obtained by applying the SE algorithm to the same problem with the new sectional running time of Train 4 on Section 1-S12. In this case, the arrived time of Train 4 at the destination reduces to **97.14** (see Figure 7-10) from 106.51 (see Figure 7-9)

To accelerate a tardy train, it seems to be essential to identify the so-called ***tardy section*** in order to make the tardy train arrive on time. In this example, for Train 4, one of the tardy sections is Section 1-S12 in Figure 7-9, which is identified according to the fact that its previous section on the routing (i.e. Section 2-S13 for inbound Train 4) has the longest blocking time.

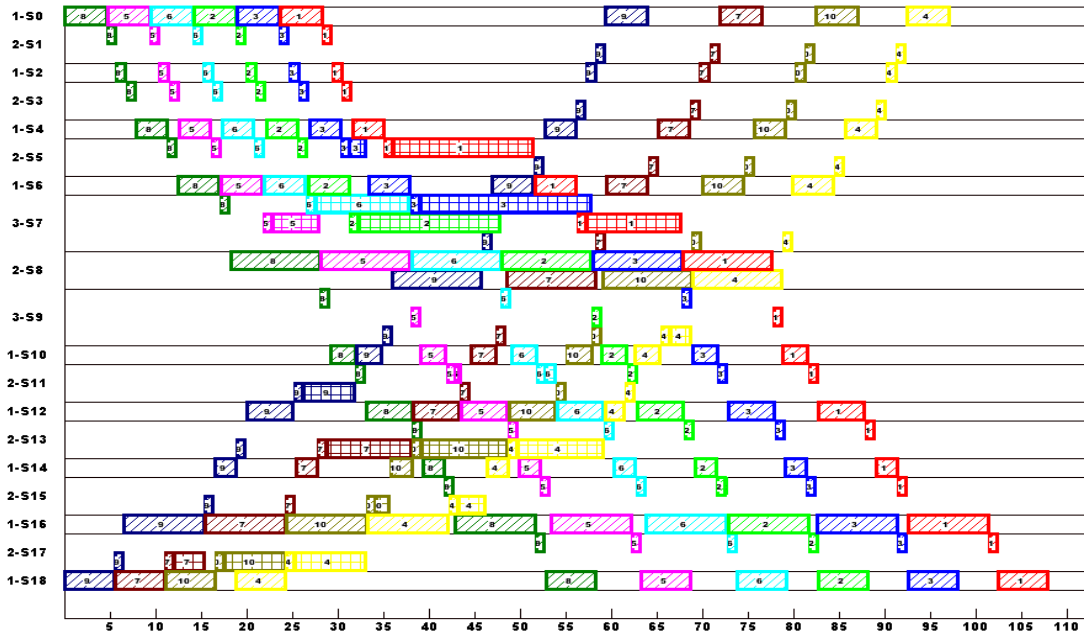

Figure 7-10: The Gantt chart of the new BPMJSS schedule when accelerating a tardy train

In practice, it is very useful in dynamic train scheduling for the tardy train to arrive early without changing the timetables of any other trains. With the above analysis, this task can be easily accomplished by simply accelerating the tardy train on an indentified tardy section.

7.2.4 Changing Bottleneck Sections

Furthermore, in many real-life situations, it is nearly impossible to upgrade the railway infrastructure of bridges or tunnels due to extremely high cost. These sections may become the so-called ***bottleneck sections***, which have the minimum gaps (or idle times) between trains. Without any investment in expanding railway facilities in these areas, it is still possible to improve the efficiency of the rail system. This phenomenon can be validated by the below results obtained when the sectional running times of all trains on a bottleneck section are shortened. See section 1-S8 in Figure 7-3 and section 1-S16 in Figure 7-11 for comparison.

The makespan is very sensitive to sectional running times on the bottleneck sections. For example, if the sectional running times of all trains on a bottleneck section (e.g. Section 1-S8 in Figure 7-3) are decreased by 50%, it is observed from the new

BPMJSS result as shown in Figure 7-11 that the new makespan drops to **110.10** from the original makespan 135.87 in Figure 7-3.

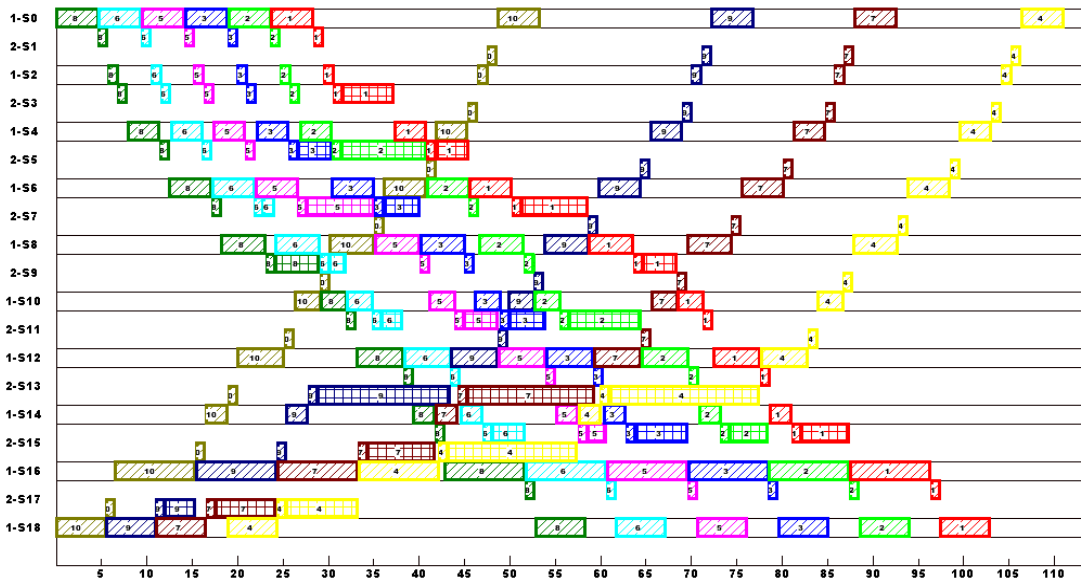

Figure 7-11: The new BPMJSS result when the sectional running times of all trains on a bottleneck section are shorten

In addition, the original bottleneck section (i.e. Section 1-S8 in Figure 7-3) is changed to be Section 1-S16 (see Figure 7-11) in this case. In practice, this implies that the cost of infrastructure upgrading can be greatly reduced by steering clear of bridges or tunnels.

7.2.5 Sensitive Analysis

Practical implementations of the proposed methodology are discussed in Sections 7.2.1-4. These applications include considering the train length, upgrading the track sections, accelerating a tardy train and changing bottleneck sections. For better comparison, the makespan values and the percentages of improvement in these applications are summarised in Table 7-2. In comparison to the original case, the percentage of improvement on the makespan of the new case is calculated by Eq.(7-1):

$$pct_{improvement} = \frac{M_{NewCase} - M_{OrginalCase}}{M_{OrginalCase}} \times 100 \qquad (7\text{-}1)$$

Table 7-2: The sensitive analysis for some practical applications in train scheduling

Cases	Makespan	Percentage of improvement (%)
Original case (Adding train length into sectional running times)	135.78	NA
New case (Excluding train length from sectional running time)	127.28	6.26
New case (Upgrading train sections)	108.03	20.44
New case (Decreasing sectional running times by 50% on a bottleneck section)	110.10	18.91
New case (Decreasing sectional running times by 40% on a bottleneck section)	118.54	14.62
New case (Decreasing sectional running times by 30% on a bottleneck section)	126.97	7.01
New case (Decreasing sectional running times by 20% on a bottleneck section)	131.46	3.35

7.3 A Case Study

To further implement and validate the proposed methodology, a case study is performed for a complex real-world coal rail system. The results obtained by the proposed methodology will be provided for decision making on building the infrastructure of a real coal rail system, which will be dedicated to the export of coal in Western Australia (Liu and Kozan, 2009a).

7.3.1 Background of Case study

Australia is the largest coal exporting country in the world. Many large coal mining operations mainly rely on the rail network to transport coal from mines to coal terminals at ports for shipment. Over the last few years, due to the fast growing demand, the coal rail network is becoming one of the worst industrial bottlenecks in Australia.

It was reported in the newspaper recently that 51 ships were queued for coal that the railway has been unable to move to the port. To be able to meet the requirements of coal export, many mining companies are willing to commit investments in building

their own coal rail network and expanding coal terminal facilities. In this context, a schematic coal railing network is drafted in Figure 7-12*.

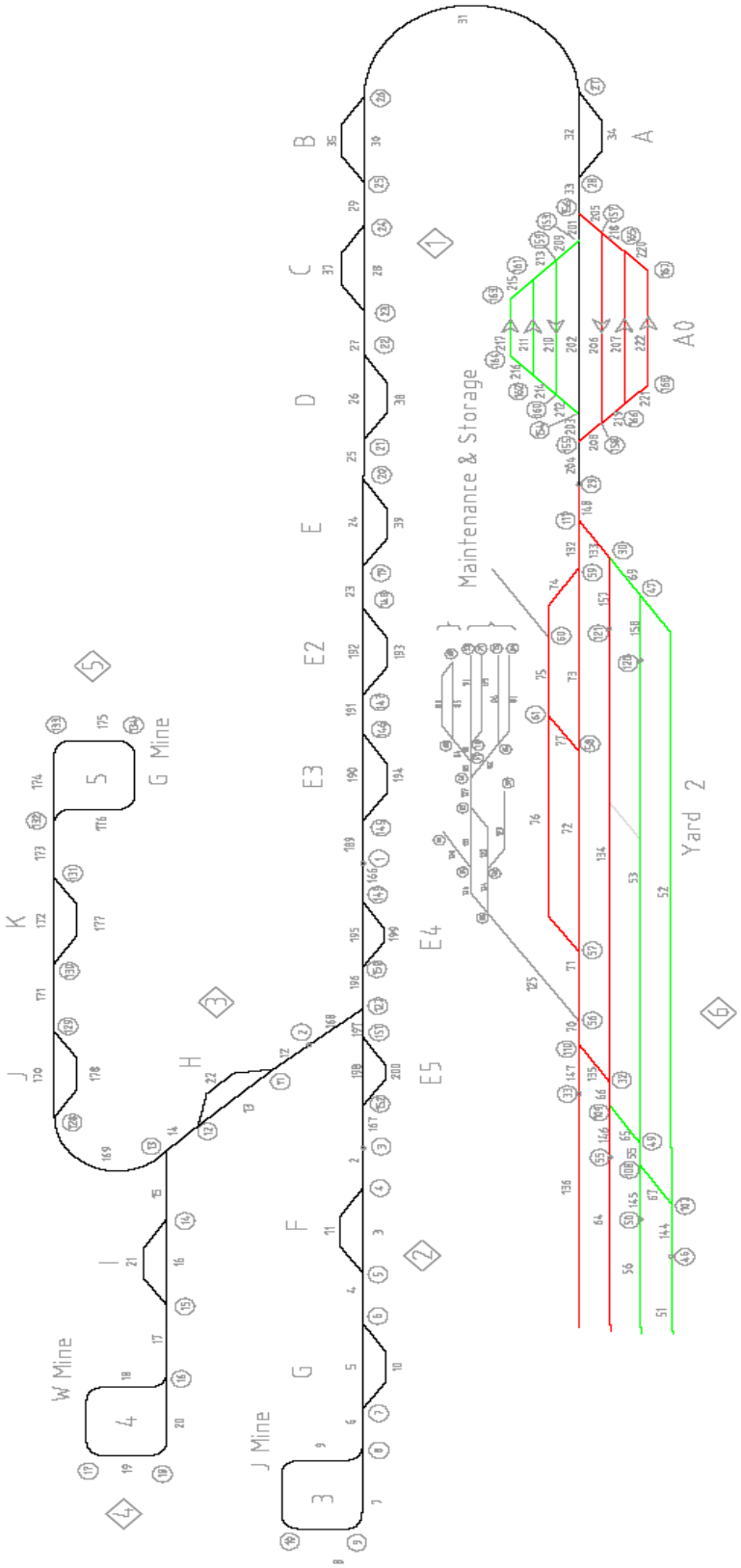

Figure 7-12: A schematic complex coal rail network

* Figure 7-12 is provided by a consultation company. For confidentiality, some parts are not shown.

The coal rail network depicted in Figure 7-12 consists of a tree-structure network of track sections that connects three mines in different locations to a coal terminal. Different types of coal in different quantities loaded from various mine sites are transported by trains to the coal terminal at port. After arriving at the coal terminal, coal is unloaded by bottom dump unloading machines and then is transferred to ships via conveyor belts.

The coal terminal has purpose-built, rail in-loading facilities, on-shore stockpile yards and off-shore wharves and shiploading equipments. The terminal acts as a limited-capacity buffer between incoming trains and outgoing ships. Thus, the coal terminal has the following two major responsibilities: receive various types of coal in various quantities from mines by trains; and deliver these various types of coal to ships.

The coal railing management system performs the following major functions:

- co-ordinating the railing of coal from various mine sites to the terminal at port;
- management and operation of loading, unloading, stockpiling and shiploading activities;
- preparation of shipping documents (timetables, quantity of coal, bills of ladings, manifests, statements of fact, etc) on behalf of mines; and
- maintenance and minor engineering functions.

Operating the coal transportation efficiently requires a series of planning and scheduling problems to be solved, including:

- ship timetabling at berth;
- stockpile management;
- train scheduling;
- loading operations at mine;
- unloading operations at stockpile yards;
- shiploading operations at port;
- rostering of crews to provide such services;

- integration for transportation in rail network and operations at coal terminal; and
- cyclic scheduling according to coal supply at mines and demand at port.

Some constraints and requirements provided from the consultation company are listed as below.

- Train configuration is measured by 4 locos and 280 wagons with the length of 3030m (i.e. 4×22.5 + 280×10.5 = 3030m).
- Loading time at mine is 4.3 hours and unloading time at terminal is 5 hours.
- Headway does not apply − only one train can traverse in a track section at a time.
- Ignore detailed yard management and apply an unloading time at the port (beyond node 29 in Figure 7-12).
- Find minimal number of trains to perform required number of services.
- Minimise the average cycle time (or the maximum completion time).
- Train timetables should be repeatable on a weekly basis.
- Transportation capacity of the rail network should be maximised.
- The number of sidings used should be minimised.
- Possible priority is for loaded trains to travel without stopping.

7.3.2 Data of Case Study

In this section, the real data of this case study are categorised and given as follows.

Firstly, according to the demand of coal export, the number of weekly services to each mine is given in Table 7-3.

Table 7-3: The number of weekly services to each mine

G Mine	J Mine	W Mine
9	24	10

Secondly, as shown in Figure 7-12, the length of each link between start node and end node is given in Table 7-4.

Table 7-4: The length of each link between start node and end node*

Link	Start Node	End Node	Length (m)	Link	Start Node	End Node	Length (m)
2	3	4	2008.947	114	89	90	140.629
3	4	5	3704.915	115	90	91	79.788
•	•	•	•	•	•	•	•
•	•	•	•	•	•	•	•
•	•	•	•	•	•	•	•
112	86	88	382.113	221	166	168	65.891
113	88	89	256.629	222	167	168	3975.627

In addition, the train speed limits in each link related to the start node are defined in Table 7-5.

Table 7-5: The train speed limit in each link with the start node*

Start Node	Link ID	Speed Limit	Start Node	Link ID	Speed Limit	Start Node	Link ID	Speed Limit	Start Node	Link ID	Speed Limit
3	2	70	26	31	69	0	137	40	131	177	70
4	2	50	27	31	66	0	138	40	128	178	70
•	•	•	•	•	•	•	•	•	•	•	•
•	•	•	•	•	•	•	•	•	•	•	•
•	•	•	•	•	•	•	•	•	•	•	•
24	28	70	0	73	40	132	173	50	0	216	25
26	30	70	0	136	40	130	177	70	0	220	25

7.3.3 Solutions of Case Study

In terms of the proposed methodology discussed in Chapter 6, the coal rail network drawn in Figure 7-12 can be modelled as a *blocking parallel job-shop scheduling* (BPMJSS) problem. This is achieved by considering each trip as jobs, which will be scheduled on single-track sections that are regarded as a set of single machines, and on double-track sections that are referred to as a set of parallel machines. As the buffer capacity between any two successive sections is zero in the train scheduling environment, the train schedule should be dead-lock free and satisfy blocking constraints.

* For confidentiality, only a portion of the data is given in Table 7-4

* Note that start node of 0 implies that the speed limit holds in both directions; and for confidentiality only a portion of the data is given in Table 7-5.

In addition, the trains that are awaiting entry to the port must be held in the seven sidings labelled A0 in Figure 7-12. This means that the port only has capacity for starting out seven trains at a time. Thus, the different earliest ready times of trains at port should be taken into account in train timetabling.

Because the sectional running times are identical for each trip between one mine site and the port, the trips can be divided into three route types:

- Route_G (i.e. the trip between G Mine and the coal terminal).
- Route_J (i.e. the trip between J Mine and the coal terminal).
- Route_W (i.e. the trip between W Mine and the coal terminal).

For one route (e.g. Route_J), one train with the current earliest ready time at port is assigned, then departs from the port, traverses and arrives at the specified mine (e.g. J Mine), loads the coal at mine, returns to the port and unloads the coal at port.

7.3.3.1 Definition of Lower Bound

The lower bound of the makespan for this case study can be defined by Eq. (7-2):

$$LB_{makepan} = \frac{\sum_{i=1}^{n_R} F_{R_i} n_{R_i}}{n_T} \tag{7-2}$$

where $LB_{makepan}$ is the lower bound of the makespan, n_R is the number of route types, R_i is the i^{th} route type, F_{R_i} is the flow time of the i^{th} route (see the definition of flow time in Section 4.2.1), n_{R_i} is the number of trips that belong to the i^{th} route type, n_T is the number of trains assigned for these trips.

The lower bound of the makespan for this case study is 135.12 calculated by Eq. (7-2):

$$LB_{makepan} = \frac{\sum_{i=1}^{n_R} F_{R_i} n_{R_i}}{n_T} = \frac{(22.01 \times 9 + 21.37 \times 24 + 23.49 \times 10)}{7} = 135.12 .$$

7.3.3.2 Initial Feasible Timetable Constructed by SE Algorithm

According to the number of weekly services to each mine given in Table 7-3, an initial sequence of trips is assumed as:

G – G – G – G – G – G – G – G – G – J – J – J – J– J – J – J – J– J – J – J – J – J – J – J – J– J – J – J – J – J – J– J – W – W – W – W – W – W– W – W – W – W,

in which G denotes Route_G, J denotes Route_J and W denotes Route_W. In total, there are 43 trips that consist of 9 Route_G trips, 24 Route_J trips and 10 Route_W trips. In terms of the above given sequence of trips, the initial feasible BPMJSS solution can be obtained by the proposed SE algorithm and displayed in Figure 7-13. The makespan of this feasible train schedule shown in Figure 7-13 is 221.44 hours. The proposed methodology is tested on a HP desktop with the Pentium IV 3.20GHz Processor and 2GB RAM. The running time of the *SE* algorithm to construct this feasible BPMJSS schedule is 50 seconds.

7.3.3.3 Good Train Schedule Found by SE-BIH Algorithm

A very good solution is found by the proposed two-stage hybrid algorithm called the *SE-BIH* algorithm (see Section 6.5) and is displayed in Figure 7-14. For this solution, the sequence of trips is:

J – G – W – J – W – G – G – J – J – W – G – W – J– J – J – W – J– G – W – J – W– J – J – J – G– W– J – J – J – G – J – J – W – J – J – G – J – J – J– W – J – G – J.

The makespan of the feasible train schedule shown in Figure 7-14 is 162.02 hours. The running time of SE-BIH algorithm to construct a very good train schedule is 17420 seconds (about 4.8 hours).

7.3.3.4 Near-Optimal Train Schedule Searched by the SE-BIH-TS Algorithm

A near-optimal (or optimal) solution is further searched using the proposed three-stage SE-BIH-TS algorithm (see Section 6.6). For this solution displayed in Figure 7-15, the sequence of trips is:

J – G – W – J – W – W – J – J – J – G – G – W– J – J – J – J– J – W – J – J– G – J – W – J– W– J – G – J – J – J – G – W – J – W – – G – J – J – J – J– G – G – W – J.

The makespan of this near-optimal feasible train schedule shown in Figure 7-15 is 160.10 hours. The running time of SE-BIH-TS algorithm with 100 TS iterations to find this near-optimal solution is 35480 seconds (about 9.8 hours).

Actually, the computation experiments indicate that the SE-TS algorithm without the BIH procedure can also yield the near-optimal solution by starting from a random initial schedule.

The comparative analysis of the results obtained by the SE-BIH-TS algorithm and the SE-TS algorithm with 100 TS iterations is given in Figure 7-16.

The analysis implies that the initial solution plays an immaterial role in TS. Therefore, to reduce the running time, it is not necessary to employ the good result obtained by SE-BIH algorithm as the initial solution for TS iterations.

7.4 Summary

In this chapter, the proposed methodology is tested and put into practice on a real-world coal rail system. The computational results validate that the proposed methodology is very promising because it can be applied to solve many real-world train scheduling problems in a reasonable time frame and provide suggestions on the improvement in the efficiency of rail systems.

From computational experiments, some conclusions can be drawn:

- In a train scheduling environment, the train length has a great impact on the performance of operating a railway. Traditionally, the *occupying time* caused by train length is ignored or added into the sectional running times. It is unsafe in real-world train scheduling to totally ignore the train length.

The second case is applicable but not precise, especially when the train length is very long. By analysis, we propose new parameters and formulae to represent the most precise and practical case.

- A section that leads to the minimum gaps (or idle times) between trains is called a *bottleneck section*. When a bottleneck section and its two adjunct sections are upgraded by retrospectively adding one more track, there is a huge improvement in railing efficiency.

- The proposed methodology is suitable for dynamic scheduling. For example, there is an easy way to make a tardy train arrive on time while the timetables of other trains are unchanged. To accomplish this task, it is critical to find the so-called *tardy section*, which is identified according to the fact that its previous section on the routing has the longest blocking time.

- The railing efficiency is very sensitive to the sectional running times on the bottleneck section. In many real-life situations, however, it is nearly impossible to upgrade the infrastructure of the bottleneck sections if they are in residential, bridge or tunnel areas. The aim of changing the bottleneck areas can be achieved by decreasing the sectional running times of all trains on the bottleneck section. As a result, the cost of expanding railway facilities can be greatly reduced by only upgrading the new bottleneck sections that are away from residential, bridge or tunnel areas.

In addition, a case study is performed for a complex real-world coal rail system. The case study shows that the proposed methodology can find the near-optimal (or optimal) feasible train schedule of a coal rail system under network and terminal capacity constraints. The obtained results can be provided for decision making on the layout of a real-world coal rail network.

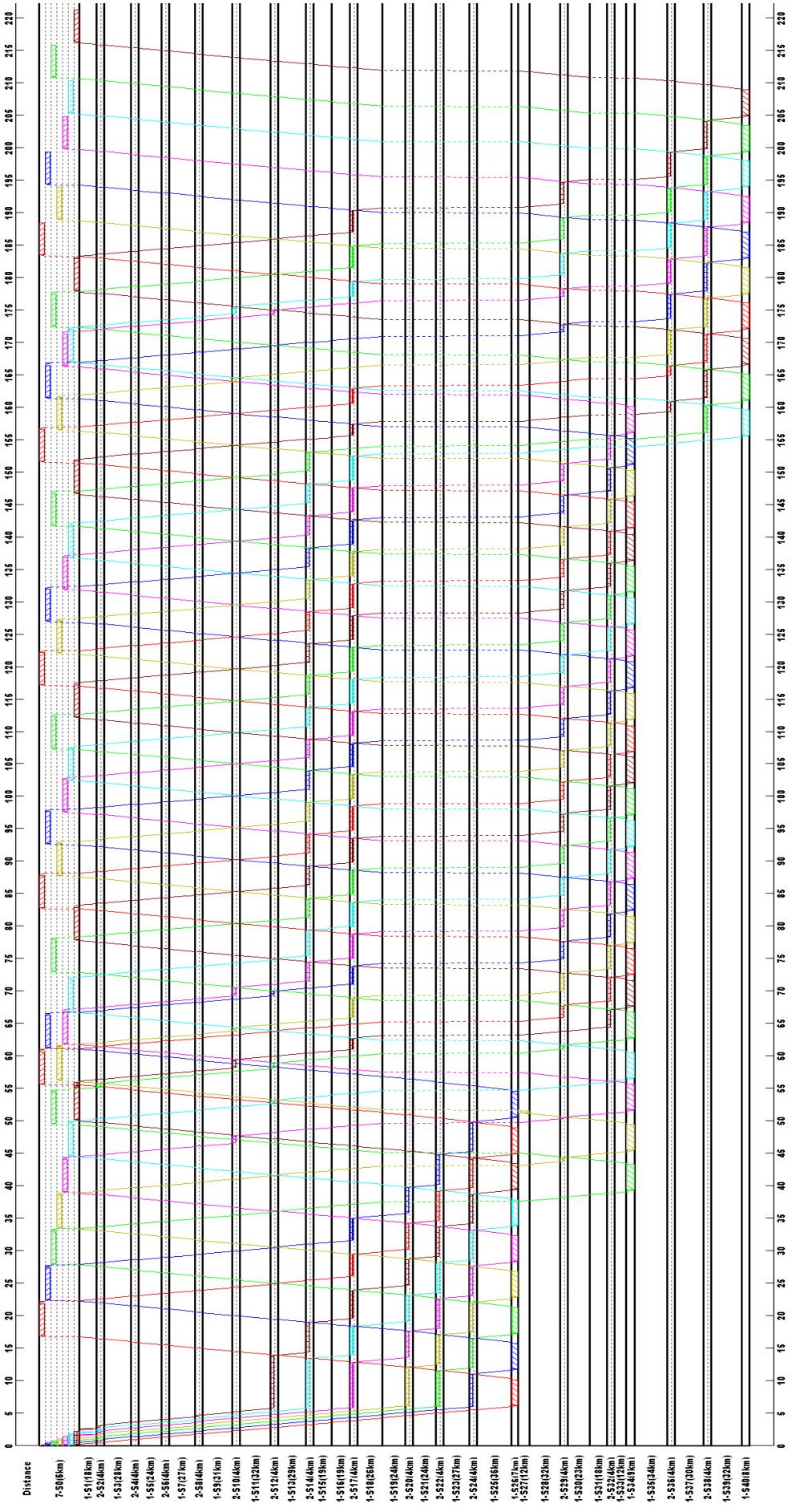

Figure 7-13: The initial result of a case study constructed by SE algorithm; note that the makespan is 221.44.

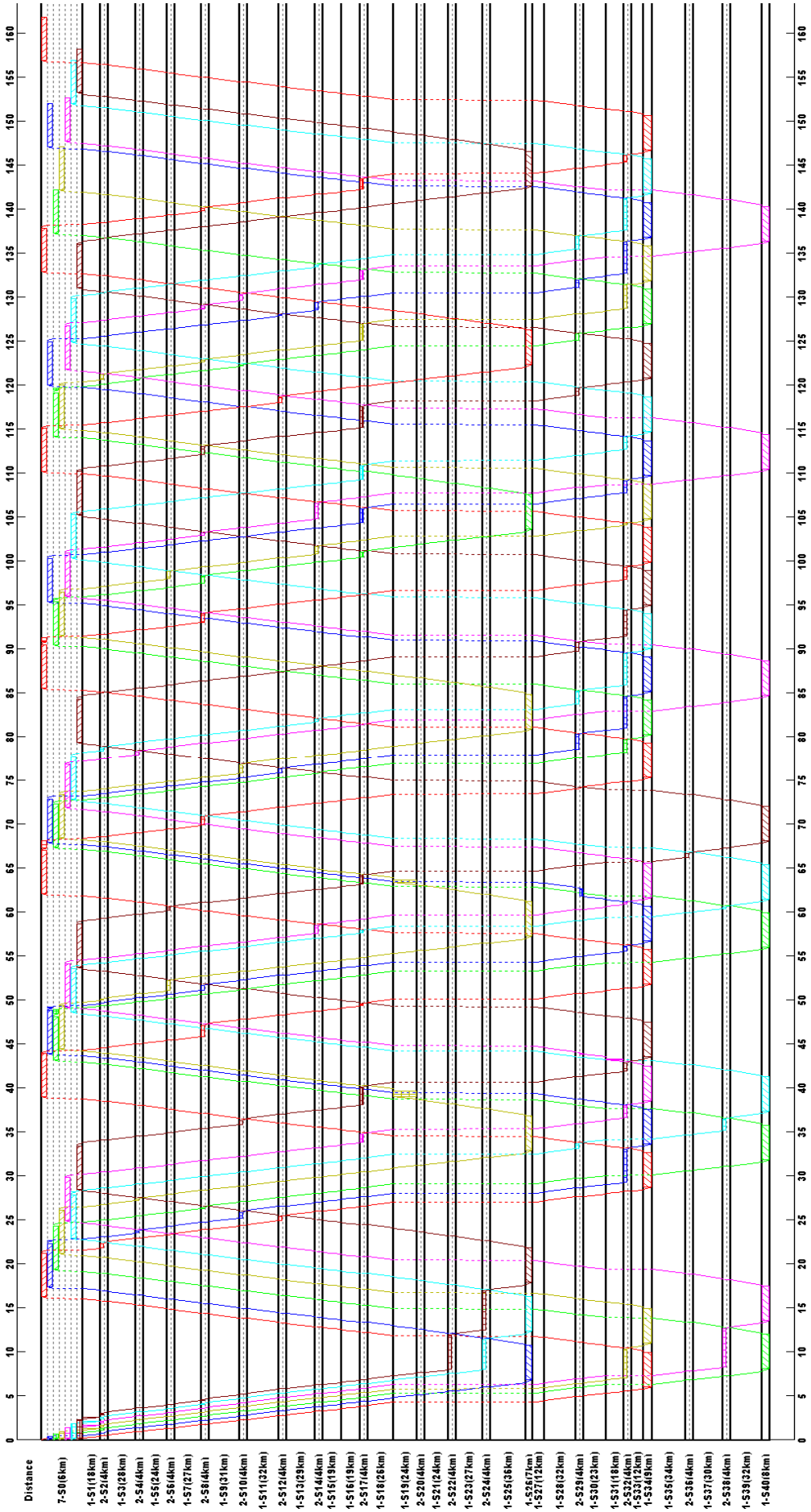

Figure 7-14: The good result of a case study obtained by SE-BIH algorithm; note that the makespan is 162.02.

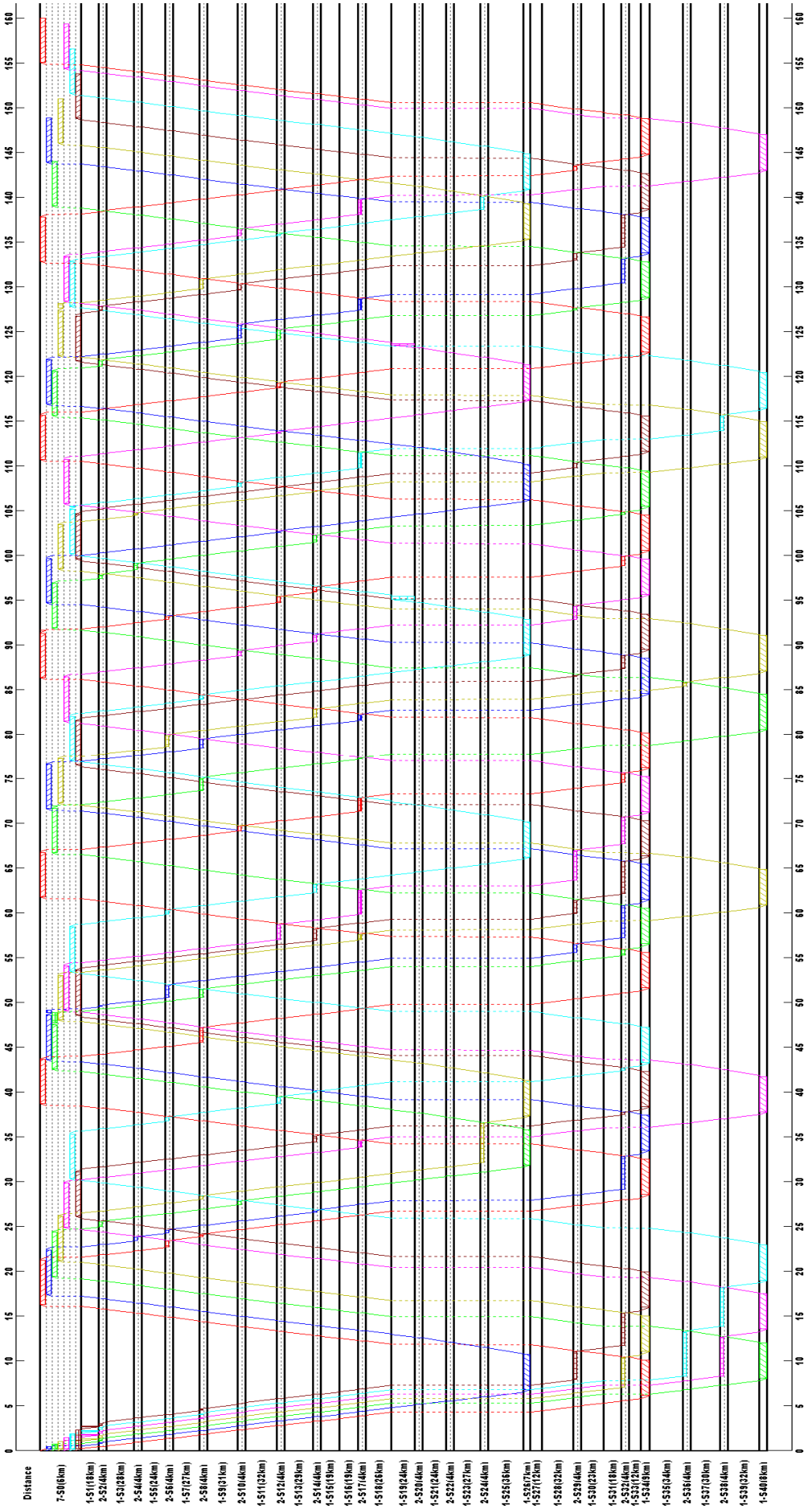

Figure 7-15: The near-optimal result of a case study obtained by SE-BIH-TS algorithm; note that the makespan is 160.10.

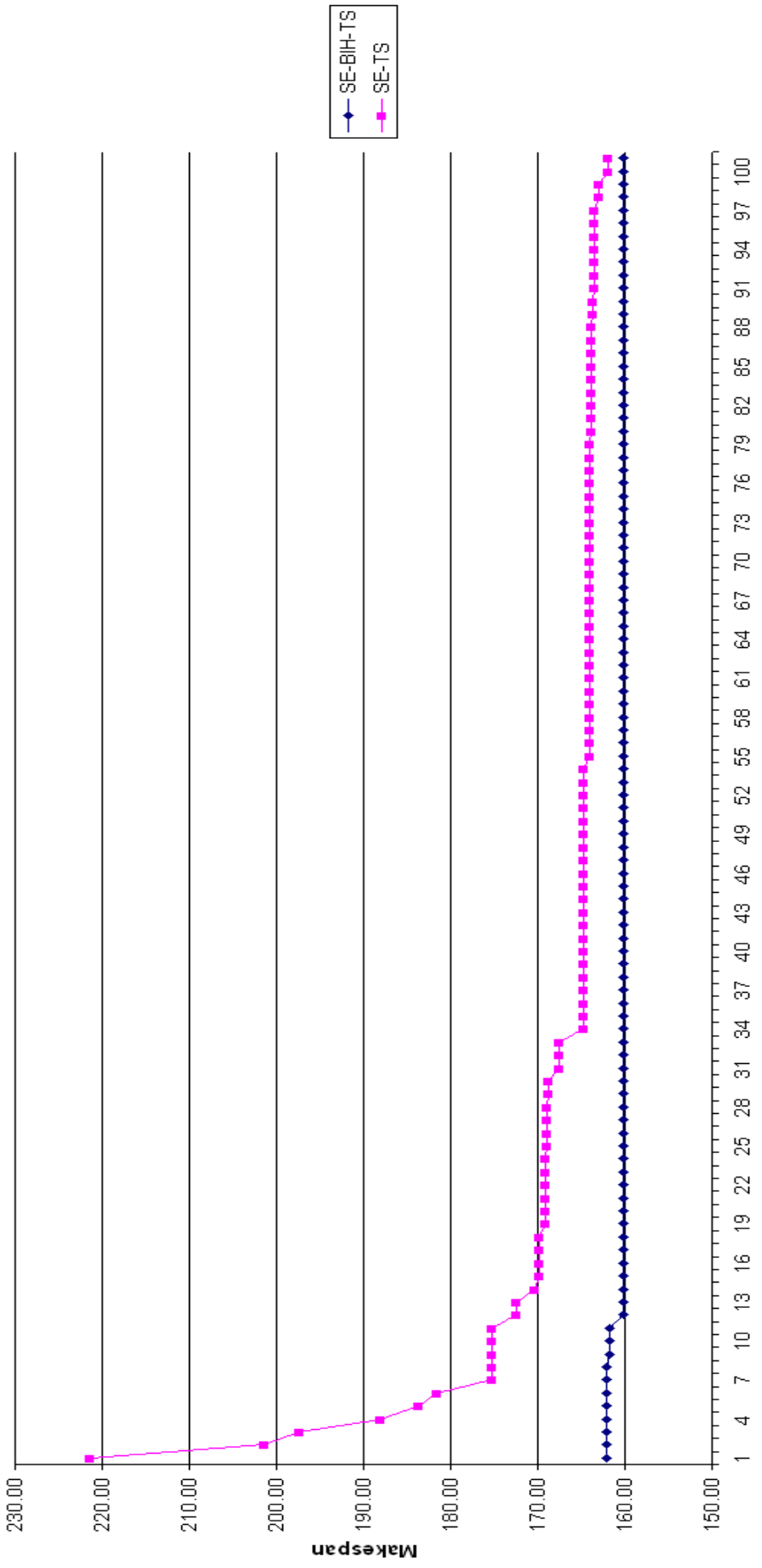

Figure 7-16: Analysis of the results obtained by the SE-BIH-TS and SE-TS algorithms with 100 TS iterations

Chapter 8 Conclusions

CHAPTER OUTLINE

8.1 Summary of Contributions

In this chapter, the main contributions of this PhD research are summarised and the directions of future research based on the outcomes presented in the previous seven chapters are given.

The aims of this PhD research are to develop standard modelling approaches and generic solution techniques for optimising a coal rail system. By achieving these goals, the followings have been completed:

- different types of train scheduling problems are modelled in terms of various real-life conditions in the rail industry;
- different solution techniques are proposed to solve these models in terms of scheduling theory;
- a scheduling software is developed using graphic interfaces; and
- the proposed methodology is implemented and verified using a real case.

During this research, extensive and fruitful studies on classical scheduling, non-classical scheduling and train scheduling problems have been done. The contributions of these research works are summarised in the following sections.

8.3.1 Contributions in Classical Scheduling

In terms of the capacity of inter-machine buffer storage, the multi-stage scheduling problems are distinguished as ***classical*** or ***non-classical***. If the capacity of buffer storage between any two successive machines is *infinite*, this problem is regarded as a *classical* type problem. Otherwise, it is referred to as a *non-classical* type problem with the *blocking*, *no-wait* or *limited-buffer* constraints.

In the literature about classical multi-stage scheduling problems, most published results are focused on the *permutation flow-shop scheduling* (PFSS), *job-shop scheduling* (JSS) and *open-shop scheduling* (OSS) problems. Comparatively, very

few articles in the literature address the *general flow-shop scheduling* (GFSS), *mixed-shop scheduling* (MSS) and *group-shop scheduling* (GSS) problems.

Four main contributions in the classical scheduling field are concluded below.

- Three metaheuristics (i.e. simulated annealing, threshed accepting and tabu search) are proposed for the PFSS and GFSS problems. Two different neighbourhood structures are used for these two types of FSS problems. From a great deal of computational experiments on the benchmark and randomly-generated instances, it is demonstrated that the optimal solution for the PFSS problem is also optimal for the corresponding GFSS result in most cases. In addition, the makespan of the best PFSS schedule is very close to the optimal makespan of the GFSS schedule.

- Furthermore, three metaheuristics were proposed for solving the FSS, JSS, OSS and MSS problems. The performance of the proposed methodology is evaluated by means of a set of Lawrence's benchmark instances for the JSS problem, a set of randomly generated instances for the OSS problem, and a combined FSS and OSS data for the MSS problem. The computational experiments show that the proposed methodology is generic and performs extremely well on solving these four types of multi-stage scheduling problems. In analysing the MSS and JSS results, it is observed that it is relatively easier to find the optimal makespan of the MSS problem in comparison with the JSS problem, due to the fact that the scheduling procedure becomes more flexible by the inclusion of the OSS-type jobs in a mixed shop.

- The GSS problem was introduced in 1997 in the context of a mathematical competition in Holland. The GSS problem is a more general formulation that covers the FSS, JSS, OSS and MSS problems. With characteristic analysis, a fast tabu search with a different neighborhood structure is developed for the GSS problem. Based on a set of benchmark GSS instances, the computational experiments show that the proposed

methodology is typically more efficient and faster than the other methods proposed in the literature. Furthermore, the new best solutions of some benchmark GSS instances can be found by the proposed tabu search.

- For the classical shop scheduling problems, most research focuses on optimizing the makespan under static conditions and does not take into consideration dynamic disturbances such as machine breakdown and new job arrivals. The shop scheduling problem under static conditions is considered as the *static shop scheduling* problem, while the shop scheduling problem with dynamic disturbances is regarded as the *dynamic shop scheduling* (DSS) problem. With exploring the characteristics of the DSS problem when machine breakdown and new job arrivals occur, an innovative framework is proposed to model the *dynamic MSS* problem as the *static GSS-type* problem in terms of the disjunctive graph. Based on the proposed framework, the methodology for the static FSS, JSS, OSS, MSS and GSS problems is successfully extended to solve the DSS problem. The computational results show that the proposed methodology is very generic and efficient in solving many classical multi-stage scheduling problems.

8.3.2 Contributions in Non-Classical Scheduling

Two main contributions in the *non-classical* scheduling field are summarised below:

- The *no-wait*, *blocking* (*no-buffer*), *limited-buffer* and *infinite-buffer* conditions for the *flow-shop scheduling* (FSS) problem have been investigated. These four different buffer conditions have been combined to generate a new class of scheduling problem, which is significant for modelling many real-world scheduling problems. For convenience, this class of non-classical scheduling problems is called the *Combined-Buffer Flow-Shop Scheduling* (CBFSS) problem, which covers the classical FSS, the blocking FSS, the no-wait FSS and the limited-buffer FSS problem. A new constructive algorithm called the *LK* algorithm, which is mainly comprised of the time-determination procedure and the tune-up procedure

with eight sets of formulae, is developed to solve this strongly NP-hard problem. Detailed numerical illustrations for the various cases are presented and analysed. The results indicate that the proposed constructive algorithm is generic for building the feasible schedule for many types of non-classical scheduling problems. The proposed approach is very promising for modelling and solving many real-world scheduling problems with different buffer constraints.

- The blocking conditions are investigated in flow-shop and job-shop environments, in which there is no buffer between any two successive machines. A new constructive algorithm called the *topological-sequence algorithm* is proposed, by exploring the properties of blocking conditions based on an *alternative graph* that is an improvement of the classical disjunctive graph. The proposed algorithm is generic and adaptive because without any modification it can be applied to solve the PFSS, GFSS and JSS problems with blocking. To indicate the superiority of the proposed algorithm, the results are compared with two well-known algorithms in the literature (i.e. *Recursive Procedure* and *Direct Graph*) only for the FSS problem with blocking.

8.3.3 Contributions in Train Scheduling

By utilising a "*toolbox*" of standard well-solved *classical* and *non-classical* scheduling problems, we develop the standard modelling approaches and generic solution techniques for a class of train scheduling problems. In real-world applications, we seek more efficient train schedules for rail transportation of coals from mines to the terminal to improve the utilisation of the whole coal railing system.

Some contributions in the train scheduling field are drawn as follows.

- An improved *shifting bottleneck procedure* (SBP) algorithm combined with metaheuristics has been developed to efficiently solve the *Parallel-Machine*

Job-Shop Scheduling (PMJSS) problems without considering the blocking conditions. Under the architecture of SBP, the topological-sequence algorithm is implemented to decompose the PMJSS problem into a set of *single-machine scheduling* (SMS) and/or *parallel-machine scheduling* (PMS) subproblems. A modified Carlier algorithm with the proposed lemmas is developed to solve the SMS subproblems. The Jackson algorithm is extended to solve the PMS subproblems.

- Due to the lack of buffer space, real-life train scheduling should consider blocking or hold-while-wait constraints, which means that a track section cannot release and must hold a train until the next section on the routing becomes available. Generally, the train scheduling problems can be modelled as a *Blocking Parallel-Machine Job-Shop Scheduling* (BPMJSS) problem. In the BPMJSS model for train scheduling, trains and sections respectively are synonymous with jobs and machines and an operation is regarded as the movement/traversal of a train across a section.

- Furthermore, the BPMJSS model is generalised to be a *No-Wait Blocking Parallel-Machine Job-Shop Scheduling* (NWBPMJSS) problem for scheduling the trains with priorities, in which prioritised trains such as express passenger trains are considered simultaneously with the non-prioritised trains like freight trains. In total, the class of train scheduling problems that have been solved include:

  - *blocking parallel-machine flow-shop-scheduling* (BPMFSS)
  - *no-wait parallel-machine flow-shop-scheduling* (NWPMFSS)
  - *blocking parallel-machine job-shop-scheduling* (BPMJSS)
  - *no-wait parallel-machine job-shop-scheduling* (NWPMJSS)
  - *no-wait blocking parallel-machine flow-shop-scheduling* (NWBPMFSS)
  - *no-wait blocking parallel-machine job-shop-scheduling* (NWBPMJSS).

- A generic algorithm called the *SE* algorithm is proposed to solve this class of train scheduling problems. To construct the feasible train timetables, the proposed algorithm consists of many individual modules including the *feasibility-satisfaction procedure*, *time-determination procedure*, *tune-up*

procedure and *conflict-resolve procedure* algorithms. To construct a good train schedule, a two-stage hybrid heuristic algorithm is developed by combining the constructive heuristic (i.e. the SE algorithm) and the local-search heuristic (i.e. the Best-Insertion-Heuristic algorithm). To optimise the train schedule, a tabu search metaheuristic is developed based on the definition of a sophisticated neighbourhood structure.

8.3.4 Contributions in Applications

Some implementations of the proposed methodology for train scheduling have been investigated and a case study has been performed for a complex real-world coal rail system.

Some main findings in the applications of the proposed methodology are summarised as follows.

- In train scheduling environments, the train length has a great impact on the performance of operating a railway. By analysis, we propose new parameters and formulae to represent the most precise and practical case.

- A section that leads to the minimum gaps (or idle times) between trains is called a *bottleneck section*. When a bottleneck sections and its two adjunct sections are upgraded by respectively adding one more track, there is a huge improvement in railing efficiency.

- The proposed methodology is suitable for dynamic scheduling. For example, there is an easy way to make a tardy train arrive on time while the timetables of other trains are unchanged. To accomplish this task, it is critical to find a *tardy section*, which is identified according to the fact that its previous section on the routing has the longest blocking time.

- The railing efficiency is very sensitive to the sectional running times on the bottleneck section. In many real-life situations, however, it is nearly

impossible to upgrade the infrastructure of the bottleneck sections if they are in bridge or tunnel areas. The aim of changing the bottleneck areas can be achieved by decreasing the sectional running times of all trains on the bottleneck section.

- A case study is performed for a complex real-world coal rail system. The results show that the proposed methodology can find the near-optimal feasible train schedule of a coal rail system under network and terminal capacity constraints. To evaluate the obtained results of this case study, a method for calculating the lower bound of the makespan is proposed. The obtained results can be provided for decision making on the infrastructure layout of a coal rail network.

In conclusion, 23 different types of scheduling problems are developed and investigated. To begin, an improved *shifting bottleneck procedure* algorithm has been combined with metaheuristics to efficiently solve the *Parallel-Machine Job-Shop Scheduling* (PMJSS) problems without considering the *blocking* conditions. Due to the lack of buffer space, real-life train scheduling should consider blocking constraints. Then, the train scheduling case can be modelled as a *Blocking Parallel-Machine Job-Shop Scheduling* (BPMJSS) problem. Furthermore, the BPMJSS model is generalised to be a *No-Wait Blocking Parallel-Machine Job-Shop Scheduling* (NWBPMJSS) problem for scheduling the trains with priorities. To develop efficient solution techniques for BPMJSS, extensive studies on the *non-classical* scheduling problems regarding the various buffer conditions (i.e. *blocking*, *no-wait*, *limited-buffer* and *combined-buffer*) have been done. With the help of characteristic analysis based on the alternative graph for non-classical scheduling problems, new algorithms are proposed for the train scheduling problems. Some real-life implementations including considering the train length, upgrading the track sections, accelerating a tardy train and changing the bottleneck sections are analysed. A case study is performed for a complex real-world coal rail system under the network and terminal capacity constraints. The outcomes show that the proposed methodology is very promising to be a fundamental tool for the real-life train scheduling problems.

8.2 Future Research

As for future research directions, the following issues need further development.

- The proposed methodology for train scheduling requires more tests before being implemented within real projects in the railway industry.

- The proposed methodology needs to be extended to the more complicated rail network with multiple lines.

- The proposed methodology needs to be developed for train scheduling problems with other criteria such as minimising the maximum number of tardy trains or the maximum tardiness, and so on.

- The proposed methodology should be extended to solve *dynamic* train scheduling problems with various dynamic factors due to the fact that unexpected events or accidents often occur in real-life train scheduling environments.

- By considering coal supplies from mines and demands from ports, it would be very useful to combine the inventory models and the proposed scheduling models in a whole coal exporting system. Thus, the class of these coal train scheduling problems are *stochastic* cases.

References

Abadi, I. N. K., Hall, N. G., & Sriskandarajah, C. (2000). Minimizing cycle time in a blocking flowshop. *Operations Research, 48*(1), 177-180.

Abdekhodaee, A., Dunstall, S., Ernst, A. T., & Lam, L. (2004). *Integration of stockyard and rail network: a scheduling case study.* Paper presented at the Proceedings of the Fifth Asia Pacific Industrial Engineering and Management Systems Conference, Gold Coast, Australia.

Adams, J., Balas, E., & Zawack, D. (1988). The shifting bottleneck procedure for job shop scheduling. *Management Science, 34*(3), 391-401.

Alcaide, D., Sicilia, J., & Vigo, D. (1997). Heuristic approaches for the minimum makespan open shop problem. *Journal of the Spanish Operational Research Society, 5*, 283-296.

Arbib, C., Italiano, G. F., & Panconesi, A. (1990). Predicting Deadlock in store-and-forward networks. *Networks, 20*, 861-881.

Baker, K. R. (1974). *Introduction to sequencing and scheduling*: New York: Wiley.

Barnes, J. W., & Chambers, J. B. (1995). Solving the job shop scheduling problem with tabu search. *IIE Transactions, 27*, 257-263.

Bellman, R. E. (1958). On a routing problem. *Quarterly Applied Mathematics, 16*, 87-90.

Brasel, H., Tautenhahn, T., & Werner, F. (1993). Constructive heuristic algorithms for the open shop. *Computing, 51*, 95-110.

Brucker, P., Hurink, J., Jurish, B., & Wostmann, B. (1997). A branch and bound algorithm for the open shop problem. *Discrete Applied Mathematics, 76*, 43-59.

Burdett, R. L., & Kozan, E. (2006). Techniques for absolute capacity determination in railways. *Transportation Research Part B, 40*, 616-632.

Cai, X., & Goh, C. J. (1994). A fast heuristic for the train scheduling problem. *Computers & Operations Research, 21*(5), 499-511.

Campbell, H. G., Dudek, R. A., & Smith, M. L. (1970). A heuristic algorithm for the n job m machine sequencing problem. *Management Science, 16*(10).

Caprara, A., Fischetti, M., & Toth, P. (2002). Modelling and solving the train timetabling problem. *Operations Research, 50*, 851-861.

Caprara, A., Monaci, M., & Toth, P. (2001). *Solution of real-world train timetabling problems*: Technical Report. University of Padova, Padova, Italy.

Caraffa, V., Ianes, S., Bagci, T. P., & Sriskandarajah, C. (2001). Minimizing makespan in a blocking flowshop using genetic algorithms. *International Journal of Production Economics, 70*, 101-115.

Carlier, J. (1982). The one-machine sequencing problem. *European Journal of Operational Research, 11*, 42-47.

Carlier, J. (1987). Scheduling jobs with release dates and tails on identical machines to minimize the makespan. *European Journal of Operational Research, 29*(3), 298-306.

Carlier, J., & Pinson, E. (1989). An algorithm for solving the job-shop problem. *Management Science, 35*(2), 164-176.

Chen, B., & Strusevich, V. A. (1993). Approximation algorithms for three-machine open shop scheduling problems. *SIAM Journal of on Computing, 5*(3), 617-632.

Chen, C. L., Neppalli, R. V., & Aljaber, N. (1996). Genetic algorithms applied to the continuous flow shop problem. *Computers & Industrial Engineering, 30*, 919-929.

Chew, K. L., Pang, J., Liu, Q. Z., Ou, J. H., & Teo, C. P. (2001). An optimization based approach to the train operator scheduling problem at Singapore MRT. *Annals of operations Research, 108*(1), 111-118.

Church, L. K., & Uzsoy, R. (1992). Analysis of periodic and event-driven rescheduling policies in dynamic shops. *International Journal of Computer Integrated Manufacturing, 5*(3), 153-163.

Dell'Amico, M., & Trubian, M. (1993). Applying tabu search to the job-shop scheduling problem. *Annals of Operations Research, 41*, 231-252.

Dirneberger, J. (2002). *Operations research and the railroad industry*: Technical Report. Montana State University, USA.

Dorfman, M. J., & Medanic, J. (2004). Scheduling trains on a railway network using a discrete event model of railway traffic. *Transportation Research Part B, 38*, 81-98.

Dueck, G., & Scheuer, T. (1990). Threshold accepting: a general purpose optimization algorithm appearing superior to simulated annealing,. *Journal of Computational Physics, 90*, 161-175.

Dutta, A. (1990). Reacting to scheduling exceptions in FMS environments. *IIE Transaction, 22*(4), 300-314.

Epstein, D. J., Lu, Q., Zhao, J., & Leachman, R. C. (2005). *An exact solution procedure for determining the optimal dispatching times for complex rail networks*. Technical report. Department of Industrial and Systems Engineering, University of Southern California, Los Angeles, CA, USA.

Fang, H. L., Ross, P., & Corne, D. (1994). *A promising hybrid GA heuristic approach for open shop scheduling problems.* Paper presented at the Proceedings of ECAI 94, 11th European Conference on Artificial Intelligence.

Fink, A., & Voß, S. (2003). Solving the continuous flow-shop scheduling problem by metaheuristics. *European Journal of Operational Research, 151*, 400-414.

Fukumori, K. (1980). *Fundamental scheme for train scheduling*: Technical report. Artificial Intelligence Laboratory, MIT.

Ghoseiri, K., Szidarovszky, F., & Asgharpour, M. J. (2004). A multi-objective train scheduling model and solution. *Transportation Research Part B, 38*, 927-952.

Gilmore, P. C., & Gomory, R. E. (1964). Sequencing a one state-variable machine: A solvable case of the travelling salesman problem. *Operations Research, 12*(5), 655-679.

Glover, F. (1986). Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research, 13*, 533-549.

Glover, F. (1989). Tabu Search-part I. *ORSA Journal on Computing, 1*(3), 190-206.

Glover, F. (1990). Tabu Search-part II. *II, ORSA Journal on Computing, 2*(1), 4-32.

Gonzalez, T., & Sahni, S. (1976). Open shop scheduling to minimize finish time. *Journal of the Association for Computing Machinery, 23*, 665-679.

Goyal, S. K., & Sriskandarajah, C. (1988). No-wait shop scheduling: computational complexity and approximate algorithms. *Operations Research, 25*, 220-244.

Grabowski, J., & Pempera, J. (2007). The permutation flow shop problem with blocking. A tabu search approach. *OMEGA, 35*, 302-311.

Grabowski, J., & Wodecki, M. (2004). A very fast tabu search algorithm for the permutation flow shop problem with makespan criterion. *Computers & Operations Research, 31*, 1891-1909.

Gupta, J. N. D. (1971). A functional heuristic algorithm for the flow shop scheduling problem. *Operational Research Quarterly, 22*(1), 61-68.

Gupta, J. N. D. (1972). Heuristic algorithms for multistage flow shop problem. *AIIE Transactions, 4*(1).

Hall, N. G., & Sriskandarajah, C. (1996). A survey of machine scheduling problems with blocking and no-wait in process. *Operations Research, 44*(3), 510-525.

Hansen, P. (1986). *The steepest ascent mildest descent heuristic for combinatorial programming.* Paper presented at the Congress on Numerical Methods in Combinatorial Optimization, Capri, Italy.

Higgins, A. (1996b). *Optimisation of Train Schedules to Minimise Transit Time and Maximise Reliability.* Queensland University of Technology, Brisbane, Australia.

Higgins, A. (1996b). *Optimisation of Train Schedules to Minimise Transit Time and Maximise Reliability.* Queensland University of Technology, Brisbane, Australia.

Higgins, A., Ferreira, L., & Kozan, E. (1995a). Modelling delay risks associated with a train schedule. *Transportation Planning and Technology, 19*(2), 89-108.

Higgins, A., Ferreira, L., & Kozan, E. (1995b). Modelling single line train operations. *Transportation Research Record, Journal of the Transportation Research Board, Railroad Transportation Research, 1489*, 9-16.

Higgins, A., Kozan, E., & Ferreira, L. (1997a). Modelling the number and location of sidings on a single line railway. *Computers & Operations Research, 24*(3), 209-220.

Higgins, A., Kozan, E., & Ferreira, L. (1997b). Heuristic techniques for single line train scheduling. *Journal of Heuristics, 3*, 43-62.

Hiller, F. S., & Lieberman, G. J. (1995). *Introduction to Operations Research*. New York, NY, USA: McGraw-Hill Publishing Company.

Holloway, C. A., & Nelson, R. T. (1974). Job shop scheduling with due dates and variable processing times. *Management Science, 20*(9), 1264-1275.

Hu, T. C. (1961). Parallel sequencing and assembly line problem. *Operations Research, 9*(6).

Ingolotti, L., Tormos, P., Lova, A., Barber, F., Salido, M. A., & Abril, M. (2005). *A decision supporting system (DSS) for the railway scheduling problem*.Unpublished manuscript, Technical Report. DSIC, Universidad Politecnica de Valencia,Spain; DEIOAC, Universidad Politecnica de Valencia, Spain; DCCIA, Universidad de Alicante, Spain.

Jackson, J. R. (1955). *Scheduling a production line to minimize maximum tardiness. Technical Report 43*. Los Angeles: University of California.

Jackson, J. R. (1955). *Scheduling a production line to minimize maximum tardiness. Technical Report 43*. Los Angeles: University of California.

Johnson, S. M. (1954). Optimal two-and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly, 1*(1).

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by Simulated Annealing. *Science, 220*(4598), 671-680.

Kozan, E., & Burdett, R. L. (2005). A railway capacity determination model and rail access charging methodologies. *Transportation Planning and Technology, 28*(1), 27-45.

Laarhoven, V., Aarts, P. J. M., & Lenstra, J. K. (1992). Job-shop scheduling by simulated annealing. *Operations Research, 22*, 629-638.

Lawrence, S. (1984). *Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques,* : GSIA, Carnegie Mellon University.

Leisten, R. (1990). Flowshop sequencing problems with limited buffer storage. *International Journal of Production Research, 28*, 2085-2100.

Leung, J. Y. T. (2004). *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Chapman & Hall/CRC.

Liaw, C. F. (1999a). Applying simulated annealing to the open shop scheduling problem. *IIE Transaction, 31*, 457-465.

Liaw, C. F. (1999b). A tabu search algorithm for the open shop scheduling problem. *Computers & operations research, 26*(2), 109-126.

Liaw, C. F. (2000). A hybrid genetic algorithm for the open shop scheduling problem. *European Journal of Operational Research, 124*(1), 28-42.

Lindner, T. (2004). *Train schedule optimization in public rail transport.* PhD thesis, der Technischen Universitat Braumschweig.

Lindner, T., & Zimmermann, U. T. (2005). *Cost Optimal Periodic Train Scheduling*. Technical Report. Institut fur Mathematische Optimierung, Technische Universitat Braunschweig.

Liu, S. Q., & Kozan, E. (2008a). Scheduling trains as a blocking parallel-machine job shop scheduling problem. *Computers and Operations Research* (In Press).

Liu, S. Q and Kozan E. (2007c). A Blocking Parallel-Machine Job-Shop-Scheduling Model for the Train Scheduling Problem. *The 8th Asia-Pacific Industrial Engineering and Management Systems Conference,* Kaohsiung, Taiwan, 10.1-10.10.

Liu, S. Q., & Kozan, E. (2007b). Scheduling a flow shop with combined buffer conditions. *International Journal of Production Economics* (In press).

Liu, S. Q., & Kozan, E. (2007a). A topological-sequence algorithm based on alternative graph for the shop scheduling problems with blocking. *Journal of Intelligent Manufacturing* (Second revision submitted).

Liu, S. Q., & Ong, H. L. (2002). A comparative study of algorithms for the flowshop scheduling problem. *Asia-Pacific Journal of Operational Research, 19*, 205-222.

Liu, S. Q., & Ong, H. L. (2004). Metaheuristics for the mixed shop scheduling problem. *Asia-Pacific Journal of Operational Research, 21*(4), 97-115.

Liu, S. Q., Ong, H. L., & Ng, K. M. (2005a). Metaheuristics for minimizing the makespan of the dynamic shop scheduling problem. *Advances in Engineering Software, 36*, 199-205.

Liu, S. Q., Ong, H. L., & Ng, K. M. (2005b). A fast tabu search algorithm for the group shop scheduling problem. *Advances in Engineering Software, 36*, 533-539.

Masics, A., & Pacciarelli, D. (2002). Job-shop scheduling with blocking and no-wait constraints. *European Journal of Operational Research, 143*, 498-517.

Masuda, T., Ishii, H., & Nishida, T. (1985). The mixed shop scheduling problem. *Discrete Applied Mathematics, 11*, 175-186.

McCormick, S. T., Pinedo, M. L., Shenker, S., & Wolf, B. (1989). Sequencing in an assembly line with blocking to minimize cycle time. *Operations Research, 37*(6), 925-935.

McNaughton, R. (1959). Scheduling with deadline and loss functions. *Management Science, 6*(1).

Mokotoff, E. (2001). Parallel machine scheduling problems: A Survey. *Asia-Pacific Journal of Operational Research, 18*(2), 193-242.

Muhleman, A. P., Lockett, A. G., & Farn, C. K. (1982). Job shop scheduling heuristics and frequency of scheduling. *International Journal of Production Research, 20*(2), 227-241.

Muntz, R. R., & Coffman, E. G. (1969). Optimal preemptive scheduling on two-processor systems. *IEEE Transactions on Computers, 18*(11).

Muth, J. F., & Thompson, G. L. (1963). *Industrial Scheduling*. Englewood Cliffs, N.J.

Nakano, R., & Yamada, T. (1991). Conventional genetic algorithm for job shop. Paper presented at *the Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo, CA, USA.

Nawaz, M., Enscore, E. E., & Ham, I. (1983). A heuristic algorithm for the m-machine, n-job flow sequencing problem. *OMEGA, 11*(1), 91-95.

Nowichi, E., & Smutnichi, C. (1996). A fast taboo search algorithm for the job-shop problem. *Management Science, 42*(6), 787-813.

Nowichi, E., & Smutnichi, C. (1996). A fast taboo search algorithm for the job-shop problem. *Management Science, 42*(6), 787-813.

Nowicki, E. (1999). The permutation flow shop with buffers: A tabu search approach. *European Journal of Operational Research, 116*, 206-219.

Ogbu, F. A., & Smith, D. K. (1990). Simulated annealing for the permutation flow shop problem. *Omega, 19*(1), 64-67.

Oliveira, E., & Smith, B. M. (2000). *A job-shop scheduling model for the single track railway scheduling problem*: Research Report Series. School of Computing, University of Leeds, UK.

Osman, I. H., & Potts, C. N. (1989). Simulated annealing for permutation flow shop scheduling. *Omega, 17*(6), 551-557.

Pacciarelli, D. (2002). The alternative graph formulation for solving complex factory scheduling problem. *International Journal of Production Economics, 40*(15), 3641-3653.

Pacciarelli, D., & Pranzo, M. (2001). *A tabu search algorithm for the railway scheduling problem.* Paper presented at the MIC'2001 - 4th Metaheuristics International Conference, 159-165.

Palmer, D. S. (1965). Sequencing jobs through a multi-stage process in the minimum total time - a quick method of obtain a near optimum. *Operational Research Quarterly, 16*(1).

Petersen, E. R. (1974). Over the road transit time for a single track railway. *Transportation Science, 8*, 65-74.

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

Prins, C. (2000). Competitive genetic algorithm for the open shop scheduling problem. *Mathematical Methods of Operations Research, 52*, 389-411.

Rajendran, C., & Chaudhuri, D. (1990). Heuristic algorithms for continuous flow-shop problem. *Naval Research Logistics, 37*, 695-705.

Ramasesh, R. (1990). Dynamic job shop scheduling: a survey of simulation research. *OMEGA, 18*(1), 43-57.

Ramudhin, A., & Marier, P. (1996). The generalized shifting bottleneck procedure. *European Journal of Operational Research, 93*(34-48).

Reddi, S. S., & Ramamoorthy, C. V. (1972). On the flow-shop sequencing problem with no wait in process. *Operational Research Quarterly, 23*(3), 323-331.

Reeves, C. R. (1993). Improving the efficiency of tabu search for machine sequencing problems. *Journal of Operational Research Society, 44*, 375-382.

Reeves, C. R. (1995). A genetic algorithm for flowshop sequencing. *Computer and Operations Research, 22*(1), 5-13.

Rinnooy, K. A. H. G. (1976). *Machine Scheduling Problems: Classification, Complexity and Computations*: The Hague: Nijhoff.

Ronconi, D. P. (2004). A note on constructive heuristics for the flowshop problem with blocking. *International Journal of Production Economics, 87*, 39-48.

Ronconi, D. P. (2005). A branch-and-bound algorithm to minimize the makespan in a flowshop with blocking. *Annals of Operations Research, 138*, 53-65.

Ronconi, D. P., & Armentano, V. A. (2001). Lower bounding schemes for flowshops with blocking in-process. *Journal of Operational Research Society, 52*, 1289-1297.

Roy, B., & Sussmann, B. (1964). *Les Problèmes d'ordonnancement avec contraintes disjonctives*. Note DS n.9 bis, SEMA, Montrouge.

Sabuncuoglu, I., & Karabuk, S. (1997). *Analysis of scheduling-rescheduling problems in a stochastic manufacturing environment*. Technical report. IEOR-9704, Department of Industrial Engineering, Bilkent University, Ankara.

Sahin, G., Ahuja, R. K., & Cunha, C. B. (2004). *New approaches for the train dispatching problem*. Technical report. Department of Industrial and Systems Engineering, University of Florida, USA.

Salido, M. A., Abril, M., Barber, F., Ingolotti, L., Tormos, P., & Lova, A. (2005). *Applying topological constraint optimization techniques to periodic train scheduling*. Technical Report. Universidad de Alicante, Spain.

Schrage, L. (1971). Obtaining optimal solutions to resource constrained network scheduling problems. Unpublished manuscript.

Shakhlevich, N. V., Sotskov, Y. N., & Werner, F. (1999). Shop-scheduling problems with fixed and non-fixed machine orders of the jobs,. *Annals of Operations Research, 92*, 281-304.

Shakhlevich, N. V., Sotskov, Y. N., & Werner, F. (2000). Complexity of mixed shop scheduling problems: a survey. *European Journal of Operational Research, 120*, 343-351.

Shi, G. (1997). A genetic algorithm applied to a classic job-shop scheduling problem. *International Journal of System Science, 28*(1), 25-32.

Storer, R. H., Wu, S. D., & Vaccari, R. (1992). New search spaces for sequencing problems with application to job shop scheduling,. *Management Science, 38*, 1495-1509.

Strusevich, V. A. (1991). Two machine super shop scheduling problem. *Journal of the Operational Research Society, 42*(6), 479-492.

Subramaniam, V., Lee, G. K., Ramesh, T., Hong, G. S., & Wong, Y. S. (2000). Machine selection rules in a dynamic job shop. *International Journal of Advanced Manufacturing Technology, 16*, 902-908.

Szelke, E., & Kerr, R. M. (1994). Knowledge-based reactive scheduling. *Production Planning and Control, 5*(2), 124-145.

Taillard, E. (1990). Some efficient heuristic methods for flow-shop sequencing. *European Journal of Operational Research, 47*, 65-74.

Taillard, E. (1993). Benchmarks for basic scheduling problems. *European Journal of Operational Research, 64*, 278-285.

Taillard, E. (1994). Parallel taboo search techniques for the job-shop scheduling problem. *ORSA Journal on Computing, 16*(2), 108-117.

The Australian. Inefficiency costs coal exports (May 30, 2007). http://www.theaustralian.news.com.au/story/0,20867,21818046-601,00.html

The Australian. Smart State's disgrace (May 30, 2007). http://www.theaustralian.news.com.au/story/0,20867,21816888-643,00.html

The Australian. Beattie keeps rail in public hands (May 30, 2007). http://www.theaustralian.news.com.au/story/0,20867,21818077-643,00.html

The Australian. Idle coal wagons infuriate industry in crisis (May 29, 2007). http://theaustralian.news.com.au/story/0,20867,21811468-2702,00.html?from=public_rss

The Australian. Idle coal wagons infuriate industry in crisis (May 29, 2007). http://theaustralian.news.com.au/story/0,20867,21811468-2702,00.html?from=public_rss

The Australian. $500m system to boost freight rail (June 17, 2008). http://www.australianit.news.com.au/story/0,,23874700-15306,00.html

Tsen, C. K. (1995). *Solving train scheduling problems using A-Teams.* PhD thesis, Carnegie Mellon University, USA.

Udomsakdigool, A., & Kachitvichyanukul, V. (2008). Multiple colony ant algorithm for job-shop scheduling problem. *International Journal of Production Research, 46*(15), 4155-4175.

Welty, G. (1991). Precision Train Scheduling? *Railway Age, 192*(8), 27-34.

Widmer, M., & Hertz, A. (1989). A new heuristic method for the flow shop sequencing problem. *European Journal of Operational Research, 41*, 186-193.

Wismer, D. A. (1972). Solution of the flowshop scheduling problem with no intermediate queues. *Operations Research, 20*, 689-697.

Yamada, T., & Nakano, R. (1996). Job-shop scheduling by simulated annealing combined with deterministic local search. *Meta-heuristics: Theory and Applications. Kluwer Academic Publishers, Hingham, MA*, 237-248.

Zhou, X., & Zhong, M. (2004). Bicriteria train scheduling for high-speed passenger railway planning applications. *European Journal of Operational Research, 167*, 752-771.

Appendix 1: A Numerical Example for NWFSS

Suppose that the permutation schedule $\Pi =< \pi_1, \pi_2, \pi_3 >=< 1,3,2 >$ of a 3-job 3-machine *no-wait flow-shop scheduling* (NWFSS) example is given, this leads to the feasible NWFSS schedule sketched in Figure A2-1 by the following calculations.

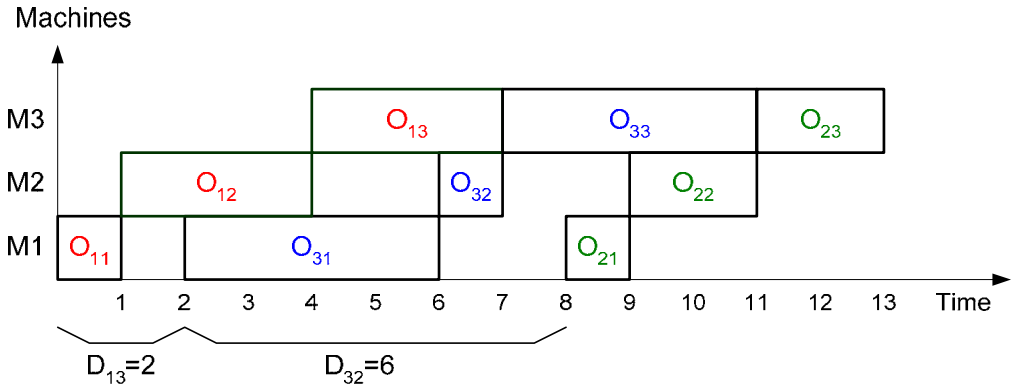

Figure A1-1: Gantt chart for illustrating a NWFSS example

As the given processing times shown in Matrix T, the delays between each two successive jobs on the first machine are calculated in Matrix D.

$$T = \begin{pmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & p_{33} \end{pmatrix} = \begin{pmatrix} 1 & 3 & 3 \\ 1 & 2 & 2 \\ 4 & 1 & 4 \end{pmatrix} \qquad D = \begin{pmatrix} - & D_{12} & D_{13} \\ D_{21} & - & D_{23} \\ D_{31} & D_{32} & - \end{pmatrix} = \begin{pmatrix} - & 4 & 2 \\ 2 & - & 1 \\ 5 & 6 & - \end{pmatrix}$$

As the given permutation schedule (i.e. $\pi =< 1,3,2 >$), the makespan is calculated by:

$$\begin{aligned} C_{\max}(\pi) &= \sum_{i=2}^{n} D_{\pi_{i-1},\pi_i} + \sum_{h=1}^{m} p_{\pi_n,h} \\ &= (D_{13} + D_{32}) + \sum_{h=1}^{m} p_{2,h} \\ &= (2+6) + (1+2+2) = 13 \end{aligned}$$

Similarly, the total flowtime is computed by:

$$\begin{aligned} F(\pi) &= \sum_{i=2}^{n} (n+1-i) D_{\pi_{i-1},\pi_i} + \sum_{i=1}^{n}\sum_{h=1}^{m} p_{ih} \\ &= (3+1-2)D_{1,3} + (3+1-3)D_{3,2} + 21 \\ &= 2\times 2 + 1\times 6 + 21 = 31 \end{aligned}$$

Appendix 2: Illustration of the Recursive-Procedure Algorithm for BFSS

For illustrating the *recursive-procedure* approach, a 4-job 3-machine *blocking flow-shop scheduling* (BFSS) example is given in Table A3-1.

Table A2-1: The processing times of a BFSS example

	M_1	M_2	M_3
J_1	$p_{11}=1$	$p_{12}=3$	$p_{13}=3$
J_2	$p_{21}=1$	$p_{22}=2$	$p_{23}=2$
J_3	$p_{31}=4$	$p_{32}=1$	$p_{33}=4$
J_4	$p_{41}=2$	$p_{42}=2$	$p_{43}=2$

Assume that the permutation schedule is $\Pi =< \pi_1, \pi_2, \pi_3, \pi_4 >=< 2,3,1,4 >$, the recursive procedure is illustrated as follows.

Step 1:

If $j=1$, as $\pi_1=2$, $D_{\pi_1,0}=D_{2,0}=0$;

On machine $k=1$:

$$D_{\pi_1,1}=D_{2,1}=\sum_{l=1}^{k}p_{2,l}=p_{21}=1$$

On machine $k=2$:

$$D_{\pi_1,1}=D_{2,2}=\sum_{l=1}^{k}p_{2,l}=p_{21}+p_{22}=1+2=3$$

On the last machine:

$$D_{2,3}=D_{2,2}+p_{2,3}=3+2=5$$

Step 2:

If $j=2$, as $\pi_j=3$ and $\pi_{j-1}=2$, $D_{\pi_j,0}=D_{3,0}=D_{\pi_{j-1},1}=D_{2,1}=1$

On machine $k=1$,

$$
\begin{aligned}
D_{\pi_j,k} = D_{3,1} &= \max\{D_{\pi_j,k-1} + p_{\pi_j,k}, D_{\pi_{j-1},k+1}\} \\
&= \max\{D_{3,0} + p_{3,1}, D_{2,2}\} \\
&= \max\{1+4, 3\} \\
&= 5
\end{aligned}
$$

On machine $k = 2$,

$$
\begin{aligned}
D_{\pi_j,k} = D_{3,2} &= \max\{D_{\pi_j,k-1} + p_{\pi_j,k}, D_{\pi_{j-1},k+1}\} \\
&= \max\{D_{3,1} + p_{3,2}, D_{2,3}\} \\
&= \max\{5+1, 5\} \\
&= 6
\end{aligned}
$$

On the last machine: $D_{3,3} = D_{3,2} + p_{3,3} = 6 + 4 = 10$

Step 3:

If $j = 3$, as $\pi_j = 1$ and $\pi_{j-1} = 3$, $D_{\pi_j,0} = D_{1,0} = D_{\pi_{j-1},1} = D_{3,1} = 5$

On machine $k = 1$:

$$
\begin{aligned}
D_{\pi_j,k} = D_{1,1} &= \max\{D_{\pi_j,k-1} + p_{\pi_j,k}, D_{\pi_{j-1},k+1}\} \\
&= \max\{D_{1,0} + p_{1,1}, D_{3,2}\} \\
&= \max\{5+1, 6\} \\
&= 6
\end{aligned}
$$

On machine $k = 2$:

$$
\begin{aligned}
D_{\pi_j,k} = D_{1,2} &= \max\{D_{\pi_j,k-1} + p_{\pi_j,k}, D_{\pi_{j-1},k+1}\} \\
&= \max\{D_{1,1} + p_{1,2}, D_{3,3}\} \\
&= \max\{6+3, 10\} \\
&= 10
\end{aligned}
$$

$$B_{1,2} = 10 - 9 = 1$$

On the last machine: $D_{1,3} = D_{1,2} + p_{1,3} = 10 + 3 = 13$

Step 4:

If $j = 4$, as $\pi_j = 4$ and $\pi_{j-1} = 1$, $D_{\pi_j,0} = D_{4,0} = D_{\pi_{j-1},1} = D_{1,1} = 6$

On machine $k = 1$:

$$
\begin{aligned}
D_{\pi_j,k} = D_{4,1} &= \max\{D_{\pi_j,k-1} + p_{\pi_j,k}, D_{\pi_{j-1},k+1}\} \\
&= \max\{D_{4,0} + p_{4,1}, D_{1,2}\} \\
&= \max\{6+2, 10\} \\
&= 10
\end{aligned}
$$

$$B_{4,1} = 10 - 8 = 2$$

On machine $k = 2$:

$$\begin{aligned} D_{\pi_j,k} = D_{4,2} &= \max\{D_{\pi_j,k-1} + p_{\pi_j,k}, D_{\pi_{j-1},k+1}\} \\ &= \max\{D_{4,1} + p_{4,2}, D_{1,3}\} \\ &= \max\{10+2, 13\} \\ &= 13 \end{aligned}$$

$$B_{4,2} = 13 - 12 = 1$$

On the last machine: $D_{4,3} = D_{4,2} + p_{4,3} = 13 + 2 = 15$

In the recursion procedure, the departure times of the first job on every machine are calculated first, then the second job, and so on until the last job. The makespan is equal to the departure time of $D_{\pi_n,m}$.

For illustration, the Gantt chart for this feasible BFSS schedule is shown in Figure A3-1. Hence, the makespan of this BFSS schedule is $D_{\pi_n,m} = D_{4,3} = 15$.

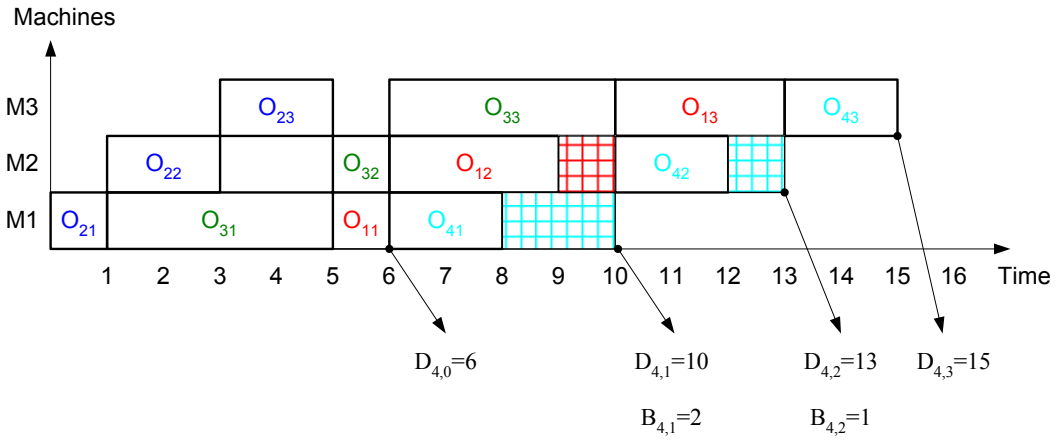

Figure A2-1: The Gantt chart for a feasible BFSS schedule

Appendix 3: Computational Experiments on the Proposed LK Algorithm

In Appendix 3, the computational experiments are conducted to validate that the proposed *LK* algorithm is generic to construct the feasible solutions for the following type scheduling problems:

- the classical *permutation flow-shop scheduling* (FSS as default) problem;
- the *blocking flow-shop scheduling* (BFSS) problem;
- the *no-wait flow-shop scheduling* (NWFSS) problem;
- the *limited-buffer flow-shop scheduling* (LBFSS) problem; and
- the *combined-buffer flow-shop scheduling* (CBFSS).

The processing times of a 5-job 5-machine CBFSS example are given in Table A1-1.

Table A3-1: The processing times of a CBFSS example

	J_1	J_2	J_3	J_4	J_5
M_1	$p_1=4$	$p_6=2$	$p_{11}=5$	$p_{16}=2$	$p_{21}=2$
M_2	$p_2=5$	$p_7=3$	$p_{12}=2$	$p_{17}=3$	$p_{22}=1$
M_3	$p_3=8$	$p_8=2$	$p_{13}=3$	$p_{18}=2$	$p_{23}=3$
M_4	$p_4=9$	$p_9=2$	$p_{14}=3$	$p_{19}=3$	$p_{24}=2$
M_5	$p_5=3$	$p_{10}=2$	$p_{15}=4$	$p_{20}=2$	$p_{25}=2$

Using a 4-field notation (See Section 3.5), this CBFSS example is denoted as:

$$F5\,|\,5\,|\,b_{12}=2, b_{23}=0, b_{34}=1, b_{45}=-1\,|\,C_{\max}\,.$$

According to the defined $b_{j,j+1}$ values in the 4-field notation, the buffer conditions between each pair of two successive machines are described as follows:

- the buffer condition before machine M_1 is: *Infinite-Buffer*;
- the buffer condition between M_1 and M_2 is: *Limited-Buffer* (2 buffers);
- the buffer condition between M_2 and M_3 is: *No-Buffer*;
- the buffer condition between M_3 and M_4 is: *Limited-Buffer* (1 buffers);
- the buffer condition between M_4 and M_5 is: *No-Wait*; and
- the buffer condition behind machine M_5 is: *Infinite-Buffer*.

After applying the *LK* algorithm to the above CBFSS example, the following feasible solution is obtained and displayed by the Gantt chart in Figure A1-1.

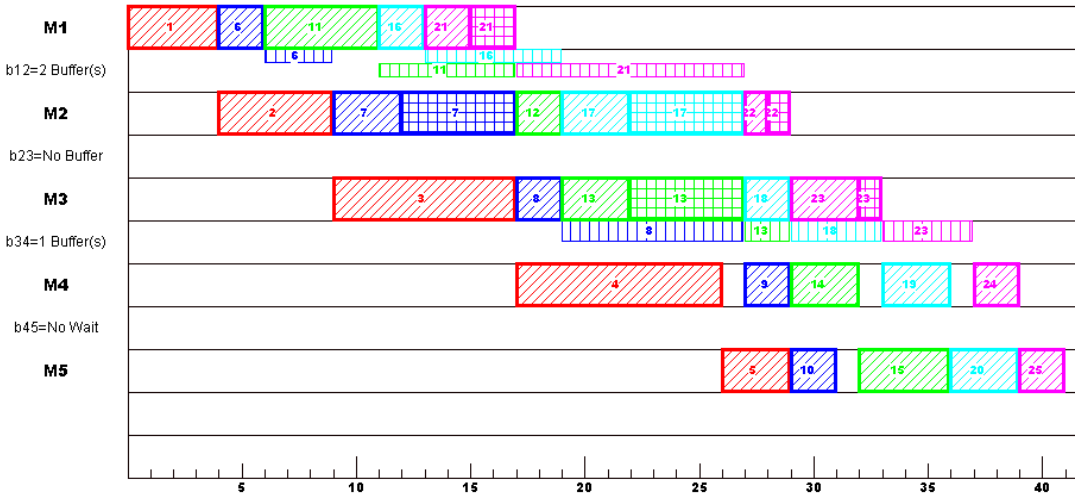

Figure A3-1: The Gantt chart for one CBFSS problem

In Figure A1-1, the box brushed by the *45 degree upward hatch* is used to indicate the starting time and finishing time of each operation on a certain machine. In addition, the box brushed by the *vertical hatch* is used to show the storing time of one operation that is stored in an earliest available buffer between two successive machines. For example, the operation 6 has to be stored from time point 6 to time point 9 in a buffer between M_1 and M_2. Moreover, if one operation blocks a machine, the blocking time is revealed by the box brushed by the *cross hatch*. For example, the operation 7, having completed on M_2 at time point 12, has to remain on M_2 till the downstream machine M_3 becomes available for processing the operation 8 (its same-job successor) at time point 17.

While keeping the same processing times and permutation schedule, the $b_{j,j+1}$ values are changed to $b_{12}=1, b_{23}=-1, b_{34}=0, b_{45}=-1$. Then, the buffer conditions between each pair of two successive machines are adjusted as follows:

- the buffer condition before machine M_1 is: *Infinite-Buffer*;
- the buffer condition between M_1 and M_2 is: *Limited-Buffer* (1 buffer);
- the buffer condition between M_2 and M_3 is: *No-Wait*;
- the buffer condition between M_3 and M_4 is: *No-Buffer*;

- the buffer condition between M_4 and M_5 is: *No-Wait*; and
- the buffer condition behind machine M_5 is: *Infinite-Buffer*.

After implementing the *LK* algorithm, the new solution for the problem with new combined buffer conditions, $F5\,|\,5\,|\,b_{12}=1, b_{23}=-1, b_{34}=0, b_{45}=-1\,|\,C_{\max}$, is displayed by the Gantt chart given in Figure A1-2. It is observed that the time points for most operations are changed for satisfying the given buffer conditions and the makespan is increased to 42.

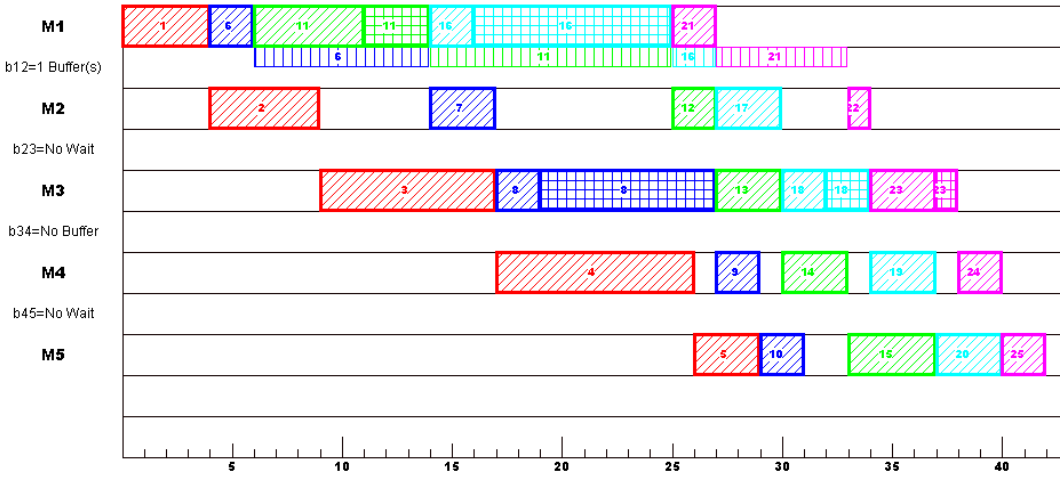

Figure A3-2: The Gantt chart for another CBFSS case

To verify that our proposed algorithm is generic, more investigations are applied to solve four extreme cases of this CBFSS example. While keeping the same processing times and permutation schedule, if all of the buffer conditions are changed to *Infinite-Buffer*, thus the problem becomes the *classical* FSS (FSS) problem. After implementing the *LK* algorithm, the new solution for this extreme case, $F5\,|\,5\,|\,b_{12}=\infty, b_{23}=\infty, b_{34}=\infty, b_{45}=\infty\,|\,C_{\max}$, is displayed in Figure A1-3. In this case, actually the calculation of blocking times and storing times of all operations can be ignored as the capacity of buffer storage is unlimited.

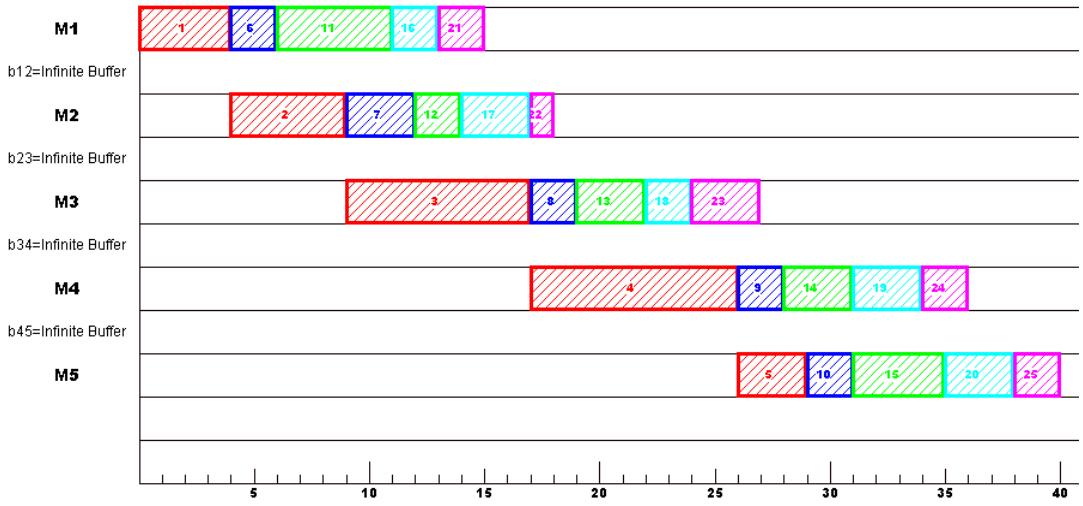

Figure A3-3: The Gantt chart for one FSS problem

If all of the buffer conditions between two successive machines are changed to *No-Buffer condition*, thus the problem is adapted to be another special case, i.e. the *blocking* FSS (BFSS) problem. After applying the *LK* algorithm, the solution for this BFSS problem, $F5 \mid 5 \mid b_{12} = 0, b_{23} = 0, b_{34} = 0, b_{45} = 0 \mid C_{\max}$, is displayed below. In this case, as there is no buffer between any two successive machines, as indicated in Figure A1-4, the storing time for every operation should be zero and none of boxes brushed by the *vertical* hatch in the Gantt chart.

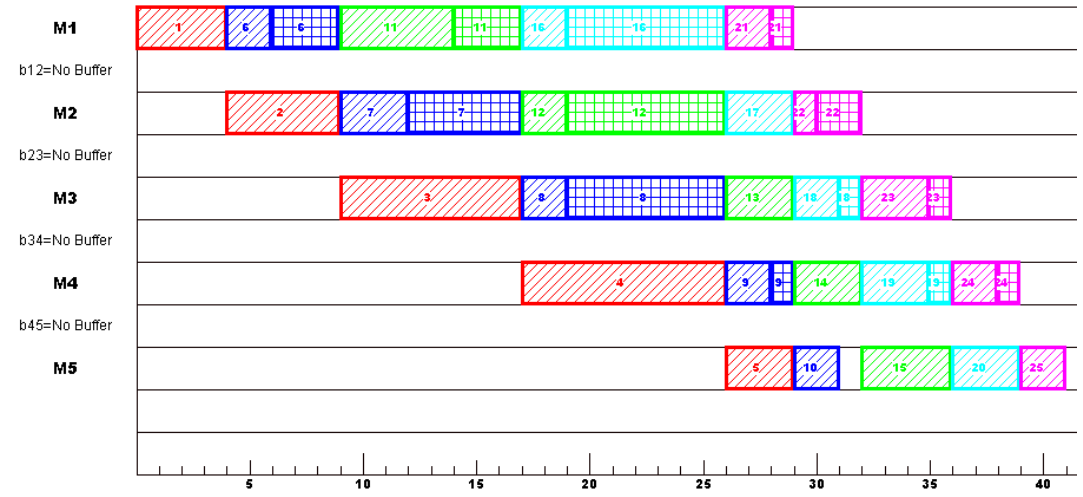

Figure A3-4: The Gantt chart for one BFSS problem

If all of the buffer conditions are changed to *Limited-Buffer*, thus the problem is interpreted as the *limited-buffer* FSS (LBFSS) problem. After implementing the *LK* algorithm, the solution for this LBFSS problem, $F5 \mid 5 \mid b_{12} = 1, b_{23} = 2, b_{34} = 3, b_{45} = 4 \mid C_{\max}$, is displayed in Figure A1-5.

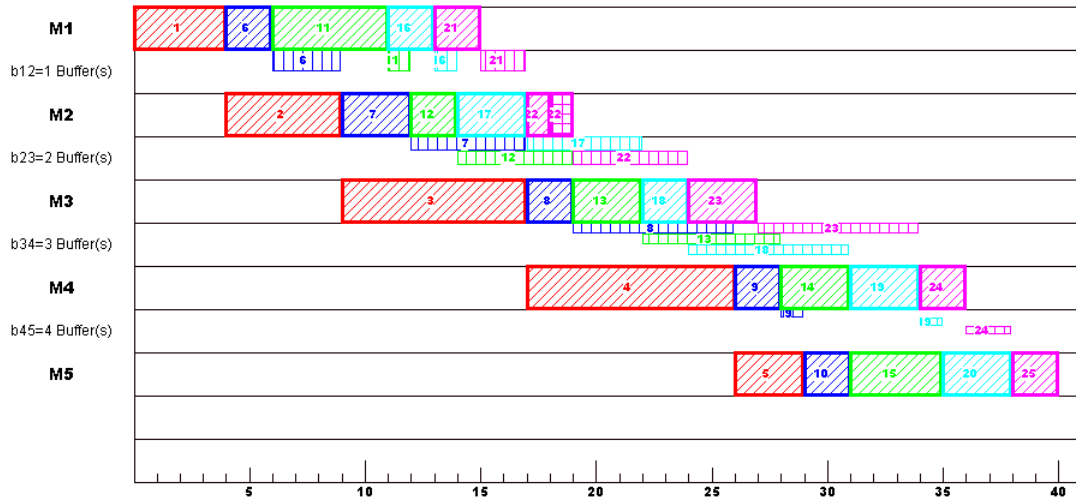

Figure A3-5: The Gantt chart for one LBFSS problem

In this case, the makespan value is reduced to 40. Except for the operation 22, there are no boxes brushed by the *cross* hatch, which means that the blocking times of nearly all operations are zero because sufficient buffer storages are available.

If all of the buffer conditions between two successive machines are changed to *No-Wait condition*, thus the problem is treated as the *no-wait* FSS (NWFSS) problem. After implementing the *LK* algorithm, the new solution for this NWFSS problem, $F5 \mid 5 \mid b_{12} = -1, b_{23} = -1, b_{34} = -1, b_{45} = -1 \mid C_{\max}$, is displayed in Figure A1-6.

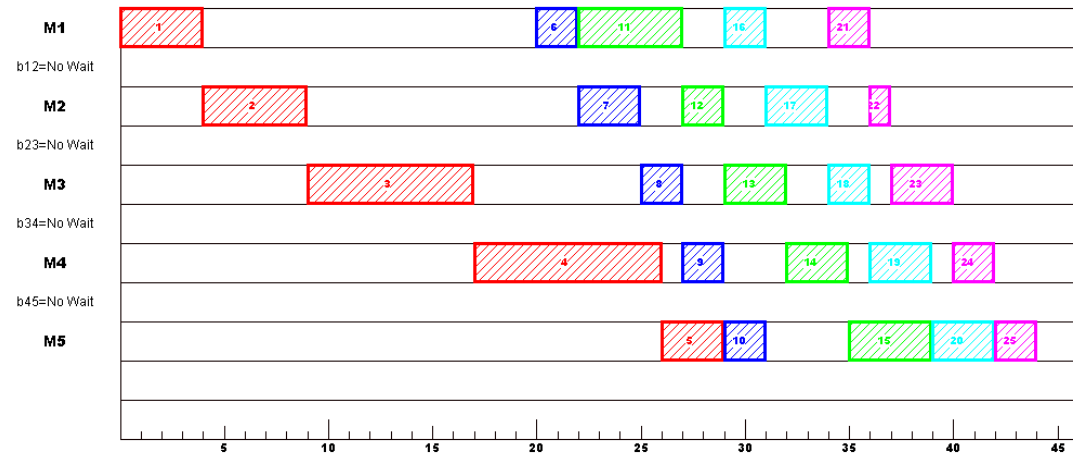

Figure A3-6: The Gantt chart for one NWFSS problem

One important observation in Figure A1-6 is that most of the jobs have to be delayed for processing on the first machine. For example, the operation 6 should start processing on the first machine at time point 20 although it has been ready at time point 0. Another observation is that, every operation's completion should

coincide with the start of its immediate same-job successor on the subsequent machine, except the last operation. Moreover, Figure A1-6 shows that the makespan value increases to 44, which demonstrates that the no-wait condition is the most restrictive constraint among these four different buffer conditions.

Appendix 4: A Numerical Example for Illustrating Schrage Algorithm

A numerical example for illustrating Schrage Algorithm is presented in Appendix 4.

The data of a 7-job *single-machine scheduling* (SMS) example with different release dates and delivering times is given in Table A4-1. This SMS example was introduced by Carlier (1982).

Table A4-1: **The data of a 7-job SMS example with release times and delivery times**

Job j	J_1	J_2	J_3	J_4	J_5	J_6	J_7
r_j (Release times)	10	13	11	20	30	0	30
p_j (Processing times)	5	6	7	4	3	6	2
q_j (Delivery times)	7	26	24	21	8	17	0

After applying the Schrage algorithm to this SMS problem, the result is shown in Figure A4-1.

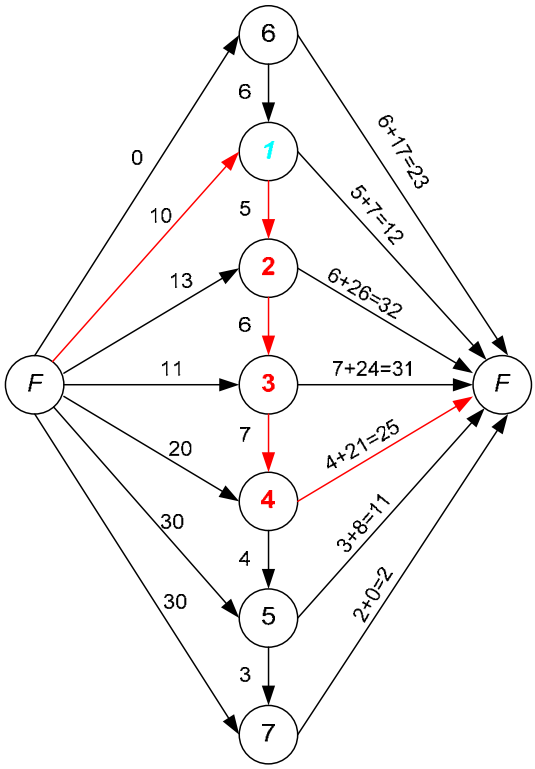

Figure A4-1: The conjunctive graph for a SMS example

The obtained SMS solution is modelled in terms of a conjunctive graph $G := \{O, A\}$. The set of nodes is obtained by adding two dummy (*First* and *Last*) nodes respectively, to the set J of jobs; that is, $O = J \bigcup \{F, L\}$.

The set A of arcs consist of three subsets: $A = A_1 \bigcup A_2 \bigcup A_3$.

- Let $A_1 = \{(F, j) \mid j \in J\}$; arc $(0, j)$ is valued by r_j, so that job j cannot start before the point at time r_j.

- Let $A_2 = \{(j, L) \mid j \in J\}$; arc (j, L) is valued by $p_j + q_j$, since job j has to spend an amount of time $p_j + q_j$ after its beginning of processing by the machine.

- Let $A_3 = \{(i, j) \mid i, j \in J\}$; arc (i, j) is valued by p_i and job i precedes job j; these arcs determine the sequence.

The schedule defined by A_3 = {6→1, 1→2, 2→3, 3→4, 4→5, 5→7}. The start-time of each node is: $e_F = 0$; $e_6 = 0$; $e_1 = 10$; $e_2 = 15$; $e_3 = 21$; $e_4 = 28$; $e_5 = 32$; $e_7 = 35$; $e_L = 53$.

Hence, the makespan is equal to 53. The critical path is $\{F, 1, 2, 3, 4, L\}$ and critical sequence is $\Lambda = \{1, 2, 3, 4\}$. The critical job c is Job 4. As $q_1 = 7$ and $q_4 = q_c = 21$, thus $q_1 < q_c$ and interference job *w* is Job 1. Thus, the critical subset is defined as $\Lambda_w = \{2, 3, 4\}$.

Appendix 5: The Proof for the Proposed Lemmas IV and V

To verify the proposed Lemmas IV and V (See Section 6.2.8), the SMS instance given in Table A4-1 is used for the following computational experiments.

Lemma IV:

For a Schrage schedule, $C_{\max} = \min_{j\in\Lambda} e_j + \sum_{j=a}^{c} p_j + \min_{j\in\Lambda} q_j$ exists.

Lemma V:

Even if $q_c = \min_{j\in\Lambda} q_j$, this Schrage schedule may not be optimal.

The following computational results will prove Lemmas IV and V.

Step 1:

After applying Schrage algorithm to this instance, we can obtain the initial results shown in Table A5-1.

Table A5-1: The initial Schrage results for one 7-job SMS problem

Results	J_1	J_2	J_3	J_4	J_5	J_6	J_7
Heads ($e_j \mid j \in J$)	10	15	21	28	32	0	35
Tails ($l_j \mid j \in J$)	38	32	25	21	8	43	0
Completion times	15	21	28	32	35	6	37
Completion-delivery times	53	53	53	53	43	49	37
Schrage Schedule	$J_6 \rightarrow J_1 \rightarrow J_2 \rightarrow J_3 \rightarrow J_4 \rightarrow J_5 \rightarrow J_7$						
Critical Sequence Λ	$J_1 \rightarrow J_2 \rightarrow J_3 \rightarrow J_4$						
Critical Job c	J_4						
Interference Job w	J_1 as $q_1 < q_4$						
Critical Subset Λ_w	$\{J_2, J_3, J_4\}$						
Makespan	53						

According to the results in Table A5-1, we observe that

$$
\begin{aligned}
C_{\max} &= \min_{j\in\Lambda} e_j + \sum_{j=a}^{c} p_j + \min_{j\in\Lambda} l_j \\
&= \min\{10,15,21,28\} + (5+6+7+4) + \min\{38,32,25,21\} \\
&= 10 + 22 + 21 \\
&= 53
\end{aligned}
$$

Step 2:

As the interference job w (Job 1) should be processed after the critical subset Λ_w = {2, 3, 4}, the release date of Job 1 is updated by:

$$
\begin{aligned}
r_w &= \max\{r_w, \min_{j\in\Lambda_w} r_j + \sum_{j\in\Lambda_w} p_j\} \\
&= \max\ \{10, \min\{13, 11, 20\}+(6+7+4)\ \} \\
&= 28
\end{aligned}
$$

After applying Schrage algorithm, the new Schrage results are shown in Table A5-2.

Table A5-2: The Schrage results after updating release date of Job 1

Results	J_1	J_2	J_3	J_4	J_5	J_6	J_7
Heads ($e_j \mid j \in J$)	28	18	11	24	33	0	36
Tails ($l_j \mid j \in J$)	11	26	32	21	8	39	0
Completion times	33	24	18	28	36	6	38
Completion-delivery times	44	50	50	49	44	45	38
Schrage Schedule	$J_6 \rightarrow J_3 \rightarrow J_2 \rightarrow J_4 \rightarrow J_1 \rightarrow J_5 \rightarrow J_7$						
Critical Sequence Λ	$J_3 \rightarrow J_2$						
Critical Job c	J_2						
Interference Job w	J_3 as $q_3 < q_2$						
Critical Subset Λ_w	$\{J_2\}$						
Makespan	50						

From Table A5-2, it is observed that

$$
\begin{aligned}
C_{\max} &= \min_{j\in\Lambda} e_j + \sum_{j=a}^{c} p_j + \min_{j\in\Lambda} l_j \\
&= \min\{18,11\} + (6+7) + \min\{26,32\} \\
&= 50
\end{aligned}
$$

Step 3:

As the interference job w (job 3) should be processed after the critical subset Λ_w = {2}, we reset the release date of Job 3 according to the formula by:

$$r_w = \max\{r_w, \min_{j\in\Lambda_w} r_j + \sum_{j\in\Lambda_w} p_j\}$$
$$= \max\{11, 13+6\} = 19$$

Then, the new Schrage results obtained by Schrage algorithm are shown in Table A5-3.

Table A5-3: The Schrage results after updating release date of Job 3

Results	J_1	J_2	J_3	J_4	J_5	J_6	J_7
Heads ($e_j \mid j \in J$)	33	13	19	26	30	0	38
Tails ($l_j \mid j \in J$)	7	32	25	21	12	38	0
Completion times	38	19	26	30	33	6	40
Completion-delivery times	45	51	51	51	45	44	30
Schrage Schedule	$J_6 \rightarrow J_2 \rightarrow J_3 \rightarrow J_4 \rightarrow J_5 \rightarrow J_1 \rightarrow J_7$						
Critical Sequence Λ	$J_2 \rightarrow J_3 \rightarrow J_4$						
Critical Job c	J_4						
Interference Job w	ϕ as $q_4 = \min_{j\in\Lambda} q_j$						
Critical Subset Λ_w	ϕ						
Makespan	51						

Lemma IV, "for a Schrage schedule, $C_{\max} = \min_{j\in\Lambda} e_j + \sum_{j=a}^{c} p_j + \min_{j\in\Lambda} q_j$ exists", is verified again by:

$$C_{\max} = \min_{j\in\Lambda} e_j + \sum_{j=a}^{c} p_j + \min_{j\in\Lambda} l_j$$
$$= \min\{13,19,26\} + (6+7+4) + \min\{32,25,21\}$$
$$= 51$$

Note that the optimal makespan is 50 with $q_3 < q_2$ ($q_2 = q_c$, $\Lambda = \{J3, J2\}$) and the current schedule satisfy $q_c \le q_j$ for all $j \in \Lambda$ but its makespan is 51. Therefore, Lemma V ("even if $q_c = \min_{j\in\Lambda} q_j$, this Schrage schedule may not be optimal") is proved. Lemma V indicates that the stopping condition (i.e. "*Schrage schedule is optimal if* $q_c = \min_{j\in\Lambda} q_j, \forall j \in \Lambda$ ") introduced in the book (Chapter 10-5, Leung 2004) may be incorrect or incomplete.

Appendix 6: The Proof for the Proposed Property 6.1

The aim of the following computational experiments is to prove Property 6.1 given in Section 6.6.2.

For a block $B_l = (\pi(u_{l-1}), \pi(u_{l-1}+1), \ldots, \pi(u_l))$, $\forall l = 1, 2, ..., k$, where it is assumed that there are k blocks in total on the critical sequence Π, we consider a subset of I-Moves $W_l(\Pi) = \{(a,b) \in V : a, b \in \{u_{l-1}+1, \ldots, u_l - 1\}\}$ which are performed insider the block B_l. Let $W(\Pi) = \bigcup_{l=1}^{k} W_l(\Pi)$, we have such an important property in designing the neighbourhood structure for the NWBPMJSS problem.

Property 6.1: For any NWBPMJSS permutation sequence $\Pi' \in N(W(\Pi), \Pi)$, we have $C_{\max}(\Pi') \geq C_{\max}(\Pi)$.

The data of a NWBPMJSS case are given in Tables A6-1, A6-2, A6-3 and A6-4.

Table A6-1: The sectional running times of a NWBPMJSS example

Machines (Sections)	Sectional Running Times (or Processing Times) of operations (in minutes)									
	J_0	J_1	J_2	J_3	J_4	J_5	J_6	J_7	J_8	J_9
M0	4.79	4.78	4.77	4.76	4.75	4.74	4.73	4.72	4.71	4.70
M1	1.02	1.02	1.02	1.02	1.02	1.03	1.03	1.03	1.03	1.03
M2	1.13	1.13	1.13	1.13	1.13	1.13	1.13	1.13	1.13	1.13
M3	1.02	1.02	1.02	1.02	1.02	1.03	1.03	1.03	1.03	1.03
M4	3.56	3.56	3.56	3.56	3.56	3.56	3.56	3.56	3.56	3.56
M5	1.02	1.02	1.02	1.02	1.02	1.03	1.03	1.03	1.03	1.03
M6	4.72	4.72	4.72	4.72	4.72	4.72	4.72	4.72	4.72	4.72
M7	1.02	1.02	1.02	1.02	1.02	1.03	1.03	1.03	1.03	1.03
M8	9.93	9.93	9.93	9.93	9.93	9.93	9.93	9.93	9.93	9.93
M9	1.02	1.02	1.02	1.02	1.02	1.03	1.03	1.03	1.03	1.03
M10	2.90	2.90	2.90	2.90	2.90	2.90	2.90	2.90	2.90	2.90
M11	1.02	1.02	1.02	1.02	1.02	1.03	1.03	1.03	1.03	1.03
M12	5.25	5.25	5.25	5.25	5.25	5.25	5.25	5.25	5.25	5.25
M13	1.02	1.02	1.02	1.02	1.02	1.03	1.03	1.03	1.03	1.03
M14	2.48	2.48	2.48	2.48	2.48	2.48	2.48	2.48	2.48	2.48
M15	1.02	1.02	1.02	1.02	1.02	1.03	1.03	1.03	1.03	1.03
M16	8.93	8.93	8.93	8.93	8.93	8.93	8.93	8.93	8.93	8.93
M17	1.02	1.02	1.02	1.02	1.02	1.03	1.03	1.03	1.03	1.03
M18	5.55	5.55	5.55	5.55	5.55	5.55	5.55	5.55	5.55	5.55

Table A6-2: **The section (machine) sequence of each train (job)**

Jobs	Section (machine) sequence of each train (job)
J_0	M0 M1 M2 M3 M4 M5 M6 M7 M8 M9 M10 M11 M12 M13 M14 M15 M16 M17 M18
J_1	M0 M1 M2 M3 M4 M5 M6 M7 M8 M9 M10 M11 M12 M13 M14 M15 M16 M17 M18
J_2	M0 M1 M2 M3 M4 M5 M6 M7 M8 M9 M10 M11 M12 M13 M15 M15 M16 M17 M18
J_3	M0 M1 M2 M3 M4 M5 M6 M7 M8 M9 M10 M11 M12 M13 M15 M15 M16 M17 M18
J_4	M0 M1 M2 M3 M4 M5 M6 M7 M8 M9 M10 M11 M12 M13 M15 M15 M16 M17 M18
J_5	M18 M17 M16 M15 M14 M13 M12 M11 M10 M9 M8 M7 M6 M5 M4 M3 M2 M1 M0
J_6	M18 M17 M16 M15 M14 M13 M12 M11 M10 M9 M8 M7 M6 M5 M4 M3 M2 M1 M0
J_7	M0 M1 M2 M3 M4 M5 M6 M7 M8 M9 M10 M11 M12 M13 M15 M15 M16 M17 M18
J_8	M18 M17 M16 M15 M14 M13 M12 M11 M10 M9 M8 M7 M6 M5 M4 M3 M2 M1 M0
J_9	M18 M17 M16 M15 M14 M13 M12 M11 M10 M9 M8 M7 M6 M5 M4 M3 M2 M1 M0

Table A6-3: The occupying times caused by train length

	J_0	J_1	J_2	J_3	J_4	J_5	J_6	J_7	J_8	J_9
Occupying time	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

Table A6-4: The definition of train directions and train priorities

	J_0	J_1	J_2	J_3	J_4	J_5	J_6	J_7	J_8	J_9
Train Direction	inbound	inbound	inbound	inbound	inbound	outbound	outbound	outbound	outbound	outbound
Train Priority	no-wait	blocking	no-wait	blocking	no-wait	blocking	no-wait	blocking	no-wait	blocking

Assume that the initial sequence of trains (jobs) for this NWBPMJSS case is:

$$\Pi = J_0 - J_1 - J_2 - J_3 - J_4 - J_5 - J_6 - J_7 - J_8 - J_9 .$$

After applying SE algorithm, the feasible NWBPMJSS timetable is obtained and displayed by the Gantt chart in Figure A6-1. The makespan of this feasible NWBPMJSS schedule is 190.84 minutes.

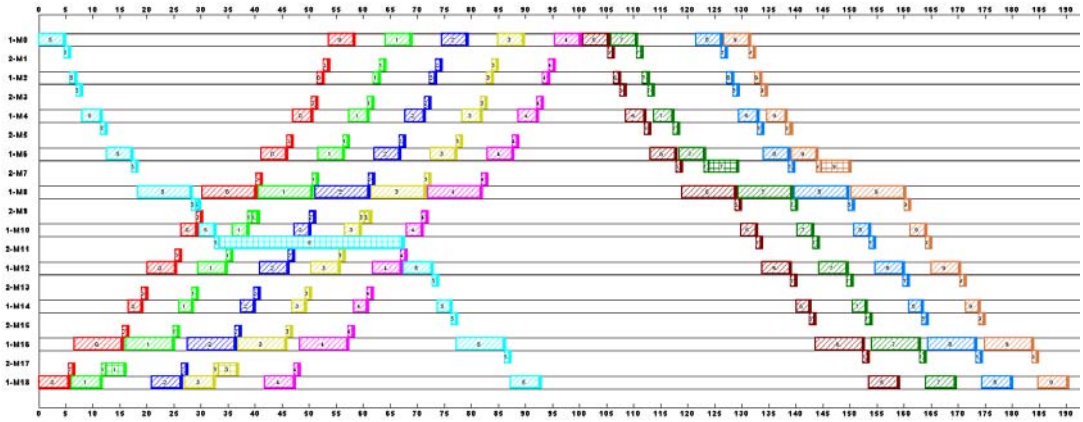

Figure A6-1: The Gantt chart for the current NWBPMJSS schedule

From Figure A6-1, the critical sequence P on a bottleneck machine (i.e.1-M8) is:

$$\mathrm{P} = (J_5 - J_0 - J_1 - J_2 - J_3 - J_4 - J_6 - J_7 - J_8 - J_9).$$

Thus, there are three blocks ($k = 3$) on the critical sequence P:

$$\mathrm{P} = (B_1, B_2, B_3)$$

$B_1 = (\pi(0))$, $B_2 = (\pi(1), \pi(2), \pi(3), \pi(4), \pi(5))$ and $B_3 = (\pi(6), \pi(7), \pi(8), \pi(9))$;

$B_1 = (J_5)$, $B_2 = (J_0, J_1, J_2, J_3, J_4)$ and $B_3 = (J_6, J_7, J_8, J_9)$.

Now we can define the set of I-moves $W(\Pi) = \bigcup_{l=1}^{k} W_l(\Pi)$ as follows:

$W_1(\Pi) = \phi$;

$W_2(\Pi) = \{(2,3), (2,4), (3,4)\}$

$W_3(\Pi) = \{(7,8)\}$

With computational experiments, the makespans of the neighbours defined by the moves $W(\Pi) = \bigcup_{l=1}^{k} W_l(\Pi)$ are shown in Table A6-5.

Table A6-5: Computational experiments for proving Property 6.1

I-Move ($W(\Pi)$)	**Neighbour** ($\Pi' \in N(W(\Pi), \Pi)$)	**Makespan of** Π'
(2,3) or (J_1,J_2)	$J_0 - J_2 - J_1 - J_3 - J_4 - J_5 - J_6 - J_7 - J_8 - J_9$	190.84
(2,4) or (J_1,J_3)	$J_0 - J_2 - J_3 - J_1 - J_4 - J_5 - J_6 - J_7 - J_8 - J_9$	190.84
(3,4) or (J_2,J_3)	$J_0 - J_1 - J_3 - J_2 - J_4 - J_5 - J_6 - J_7 - J_8 - J_9$	190.84
(7,8) or (J_7,J_8)	$J_0 - J_1 - J_2 - J_3 - J_4 - J_5 - J_6 - J_8 - J_7 - J_9$	190.84

From Table A6-5, it is proved that any NWBPMJSS neighbouring solution in the neighbourhood defined by $N(W(\Pi), \Pi)$ definitely cannot improve the solution quality due to $C_{\max}(\Pi') \geq C_{\max}(\Pi)$, $\forall \Pi' \in N(W(\Pi), \Pi)$.

Appendix 7: The Performance of the Proposed Neighbourhood Structure

The computation experiments for the performance of the proposed neighbourhood structure $N(Z(\Pi),\Pi)$ are given below.

The example is the same as given in Appendix 6. In addition, the current permutation sequence of trains is also same:

$$\Pi = J_0 - J_1 - J_2 - J_3 - J_4 - J_5 - J_6 - J_7 - J_8 - J_9 .$$

Thus, the same feasible NWBPMJSS timetable obtained by SE algorithm is displayed in Figure A6-1. Repeatedly, the critical sequence P on a bottleneck machine (i.e.1-M8 in Figure A6-1) is:

$$\mathrm{P} = (J_5 - J_0 - J_1 - J_2 - J_3 - J_4 - J_6 - J_7 - J_8 - J_9) .$$

Thus, there are three blocks ($k = 3$) on the critical sequence P:

$$\mathrm{P} = (B_1, B_2, B_3)$$

$B_1 = (\pi(0))$, $B_2 = (\pi(1),\pi(2),\pi(3),\pi(4),\pi(5))$ and $B_3 = (\pi(6),\pi(7),\pi(8),\pi(9))$;

$B_1 = (J_5)$, $B_2 = (J_0, J_1, J_2, J_3, J_4)$ and $B_3 = (J_6, J_7, J_8, J_9)$.

The makespan of the current NWBPMJSS schedule is 190.84.

Now such a set of concise moves $Z(\Pi) = \bigcup_{l=1}^{k}[ZR_l(\Pi) \bigcup ZL_l(\Pi)]$ can be defined below.

$ZL_1(\Pi) = \phi$

$ZL_2(\Pi) = \{(2,1),(3,1),(4,1),(5,1)\}$

$ZL_3(\Pi) = \{(7,6),(8,6),(9,6)\}$

$ZR_1(\Pi) = \phi$

$ZR_2(\Pi) = \{(1,5),(2,5),(3,5),(4,5)\}$

$ZR_3(\Pi) = \{(6,9),(7,9),(8,9)\}$

The computational experiments for analysing the neighbourhood are conducted by the moves $Z(\Pi) = \bigcup_{l=1}^{k}[ZR_l(\Pi) \bigcup ZL_l(\Pi)]$ and summarised in Table A7-1.

Table A7-1: Computational experiments for analysing the proposed neighbourhood structure

I-Move ($Z(\Pi)$)	**Neighbour** ($\Pi' \in N(Z(\Pi), \Pi)$)	**Makespan of** Π'
(2,1) or (J_1,J_0)	$J_1 - J_0 - J_2 - J_3 - J_4 - J_5 - J_6 - J_7 - J_8 - J_9$	190.84
(3,1) or (J_2,J_0)	$J_2 - J_0 - J_1 - J_3 - J_4 - J_5 - J_6 - J_7 - J_8 - J_9$	190.84
(4,1) or (J_3,J_0)	$J_3 - J_0 - J_1 - J_2 - J_4 - J_5 - J_6 - J_7 - J_8 - J_9$	190.84
(5,1) or (J_4,J_0)	$J_4 - J_0 - J_1 - J_2 - J_3 - J_5 - J_6 - J_7 - J_8 - J_9$	190.85
(7,6) or (J_7,J_6)	$J_0 - J_1 - J_2 - J_3 - J_4 - J_5 - J_7 - J_6 - J_8 - J_9$	180.41
(8,6) or (J_8,J_6)	$J_0 - J_1 - J_2 - J_3 - J_4 - J_5 - J_8 - J_6 - J_7 - J_9$	190.82
(9,6) or (J_9,J_6)	$J_0 - J_1 - J_2 - J_3 - J_4 - J_5 - J_9 - J_6 - J_7 - J_8$	180.41
(1,5) or (J_0,J_4)	$J_1 - J_2 - J_3 - J_4 - J_0 - J_5 - J_6 - J_7 - J_8 - J_9$	190.88
(2,5) or (J_1,J_4)	$J_0 - J_2 - J_3 - J_4 - J_1 - J_5 - J_6 - J_7 - J_8 - J_9$	190.87
(3,5) or (J_2,J_4)	$J_0 - J_1 - J_3 - J_4 - J_2 - J_5 - J_6 - J_7 - J_8 - J_9$	190.86
(4,5) or (J_3,J_4)	$J_0 - J_1 - J_2 - J_4 - J_3 - J_5 - J_6 - J_7 - J_8 - J_9$	190.85
(6,9) or (J_6,J_9)	$J_0 - J_1 - J_2 - J_3 - J_4 - J_5 - J_7 - J_8 - J_9 - J_6$	180.39
(7,9) or (J_7,J_9)	$J_0 - J_1 - J_2 - J_3 - J_4 - J_5 - J_6 - J_8 - J_9 - J_7$	190.84
(8,9) or (J_8,J_9)	$J_0 - J_1 - J_2 - J_3 - J_4 - J_5 - J_6 - J_7 - J_9 - J_8$	190.84

From Table A7-1, it is observed that the solution quality can be improved from the neighbourhood constructed by the set of moves $Z(\Pi) = \bigcup_{l=1}^{k}[ZR_l(\Pi) \bigcup ZL_l(\Pi)]$, because the makespan value of 180.39 is found among the neighbouring solutions by I-Move (6,9) at the first TS iteration.

The proposed TS algorithm is applied step by step and the best NWBPMJSS schedule among the neighbourhood at each TS iteration is displayed by the Gantt chart, as shown in Figures A7-1, A7-2 and A7-3.

TS Iteration 1:

At the first iteration of TS algorithm, the new solution is shown in Figure A7-1. It is observed that the new makespan reduces to be **180.39** with a new sequence of trains obtained as: {J_0, J_1, J_2, J_3, J_4, J_5, J_7, J_6, J_8, J_9}.

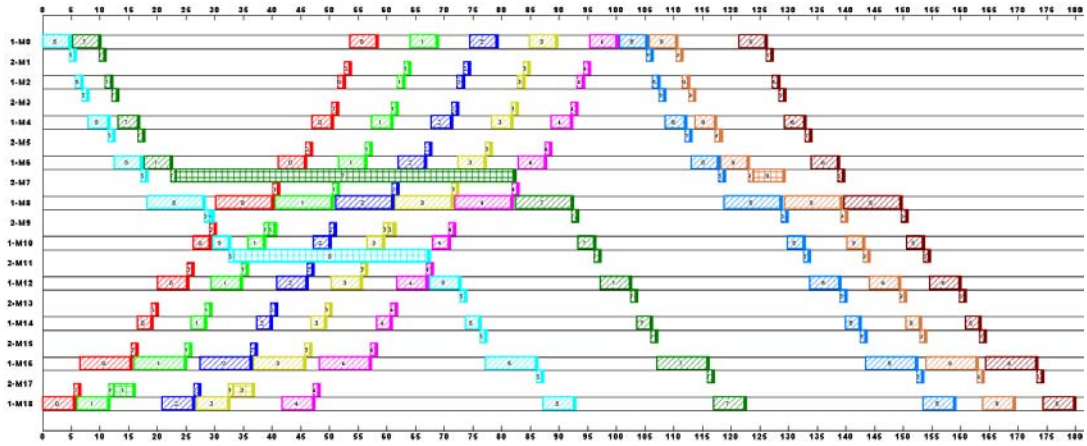

Figure A7-1: Gantt chart for the best schedule found at the first iteration of TS

TS Iteration 2:

At the second iteration of TS algorithm, the makespan of the new solution decreases to **169.96** with the new sequence of trains: $\{J_0, J_1, J_2, J_3, J_4, J_5, J_7, J_9, J_6, J_8\}$. The best result at the second TS iteration is shown in Figure A7-2.

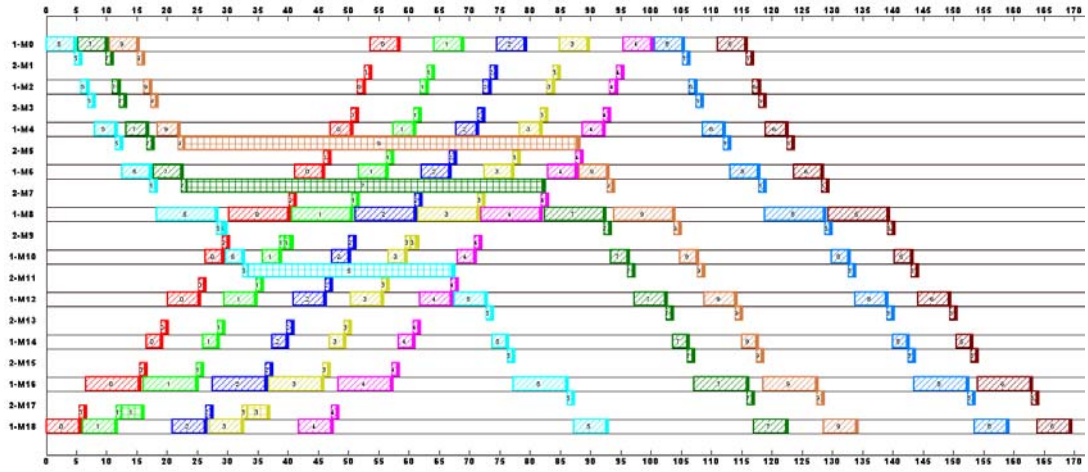

Figure A7-2: Gantt chart for the best schedule found at the second iteration of TS

TS Iteration 3:

At the third iteration of TS, the makespan of the new solution continuously decreases to **165.37** and the new sequence of trains becomes $\{J_0, J_1, J_3, J_4, J_5, J_7, J_2, J_9, J_6, J_8\}$ and is displayed in Figure A7-3.

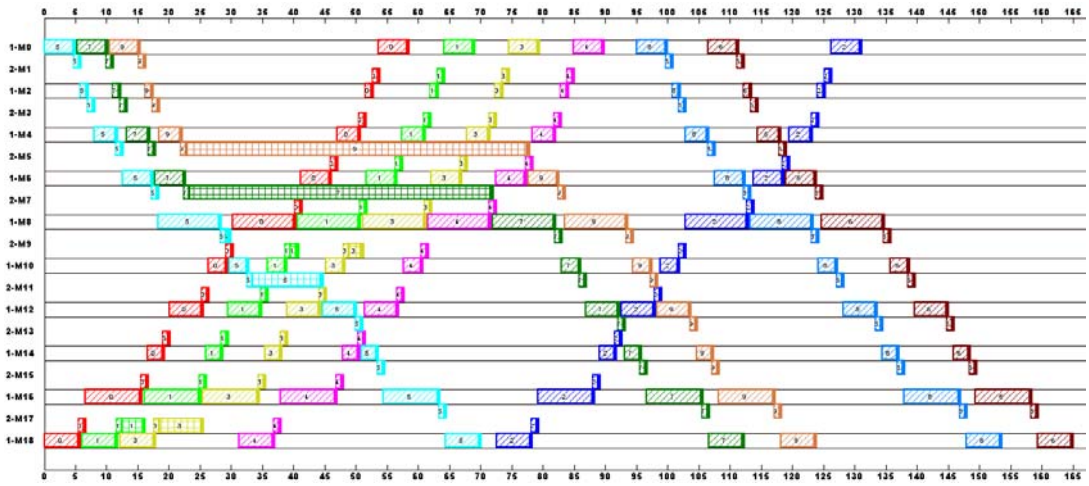

Figure A7-3: Gantt chart for the improved schedule at the third iteration of TS

To sum up, it is observed that the makespan of this NWBPMJSS example can be decreased consecutively in the first three iterations of TS from190.94 to 180.39 and 169.96 and finally to 165.37.

The computational experiments sufficiently prove that the proposed neighbourhood structure is efficient and effective to guide the neighbourhood search procedure.

Appendix 8: Computational Experiments on the Proposed SE Algorithm for Train Scheduling Problems

Appendix 8 will verify that the proposed SE algorithm is very generic because it is able to construct the feasible timetables for the following six train scheduling problems by only adjusting the attributes in the data input.

In terms of various settings of train directions (outbound or inbound) and train priorities (freight or passenger), these six train scheduling problems are classified as follows.

1. *Blocking Parallel-Machine Flow-Shop-Scheduling* (BPMFSS)
2. *Blocking Parallel-Machine Job-Shop-Scheduling* (BPMJSS)
3. *No-Wait Parallel-Machine Flow-Shop-Scheduling* (NWPMFSS)
4. *No-Wait Blocking Parallel-Machine Flow-Shop-Scheduling* (NWBPMFSS)
5. *No-Wait Parallel-Machine Job-Shop-Scheduling (*NWPMJSS)
6. *No-Wait Blocking Parallel-Machine Job-Shop-Scheduling* (NWBPMJSS)

For convenience, the test instance given in Appendix 6 is employed for the following computational experiments. The data in Tables A6-1, A6-2, A6-3 are unchanged, but the data in Table A6-4 about train directions and train priorities are updated every time for representing different cases. In addition, the permutation sequence of trains for all cases is kept the same as: $\{T_0, T_1, T_2, T_3, T_4, T_5, T_7, T_6, T_8, T_9\}$.

Case 1: BPMFSS

In Case 1, the data about train directions and train priorities are defined in Table A8-1. In this case, the directions of all trains are *outbound* and the priorities of all trains are *blocking*. Thus, this case is a BPMFSS problem.

Table A8-1: The definition of train directions and train priorities for a BPMFSS case

	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
Train Direction	outbound	outbound	outbound	outbound	outbound	outbound	outbound	outbound	outbound	outbound
Train Priority	blocking	blocking	blocking	blocking	blocking	blocking	blocking	blocking	blocking	blocking

By applying the proposed SE algorithm, the result of case 1 is obtained and shown in Figure A8-1, in which the boxes of blocking times are highlighted by the cross brush.

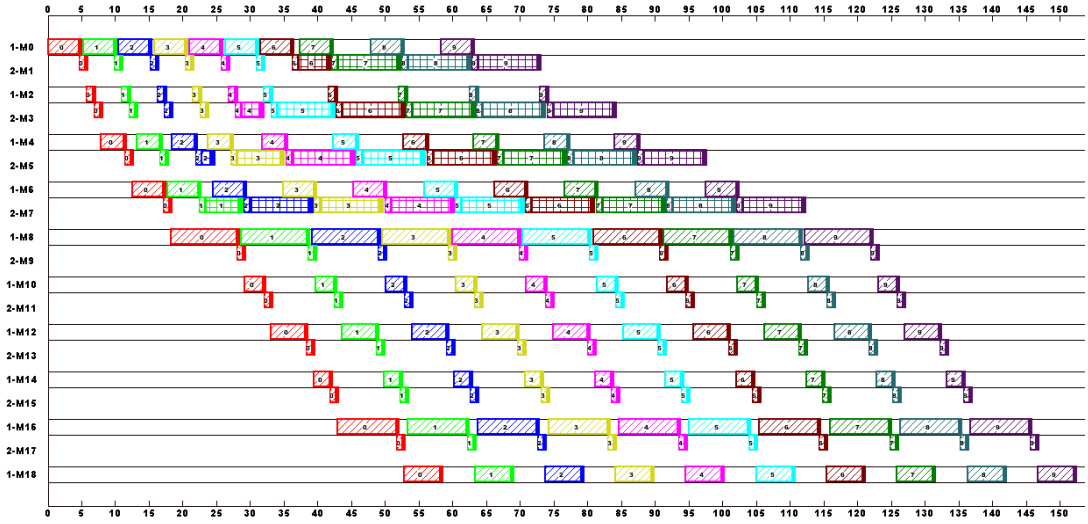

Figure A8-1: The Gantt chart for a BPMFSS case

Case 2: BPMJSS

In Case 2, the data about train directions and train priorities are updated in Table A8-2.

Table A8-2: The definition of train directions and train priorities for a BPMJSS case

	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
Train Direction	inbound	inbound	inbound	inbound	inbound	outbound	outbound	outbound	outbound	outbound
Train Priority	blocking	blocking	blocking	blocking	blocking	blocking	blocking	blocking	blocking	blocking

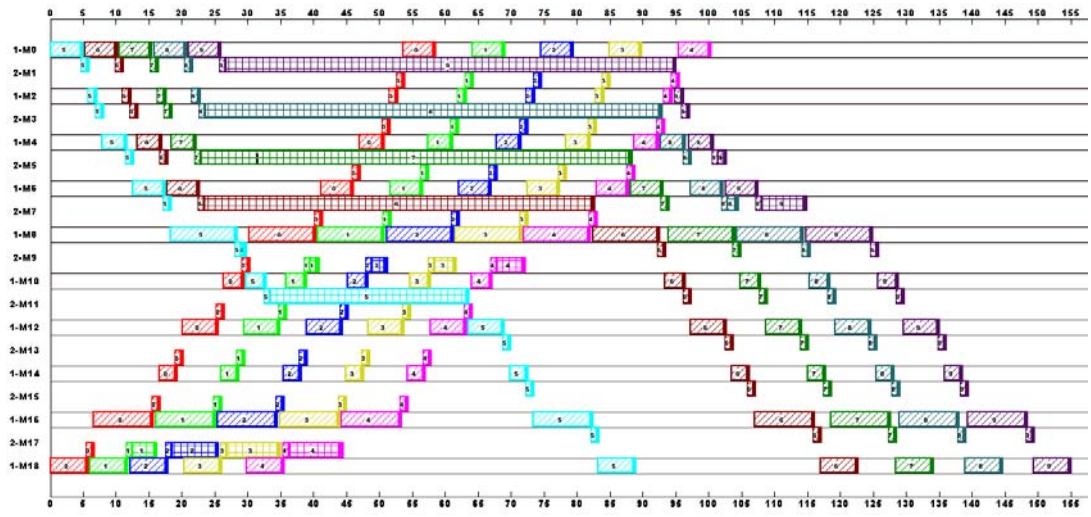

Figure A8-2: The Gantt chart for a BPMJSS case

In this case, the directions of some trains are *inbound* while the directions are *outbound* for the other trains. Thus, the problem type is changed to be BPMJSS. After applying the SE algorithm, the feasible BPMJSS solution is shown in Figure A8-2.

Case 3: NWPMFSS

In Case 3, the data about train directions and train priorities are defined in Table A8-3. In this case, the directions of all trains are *inbound* and the priorities of all trains are *no-wait*. Thus, the problem type is changed to be NWPMFSS. After applying the SE algorithm, the feasible NWPMFSS solution is shown in Figure A8-3.

Table A8-3: The definition of train directions and train priorities for a NWPMFSS case

	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
Train Direction	inbound	inbound	inbound	inbound	inbound	inbound	inbound	inbound	inbound	inbound
Train Priority	no-wait	no-wait	no-wait	no-wait	no-wait	no-wait	no-wait	no-wait	no-wait	no-wait

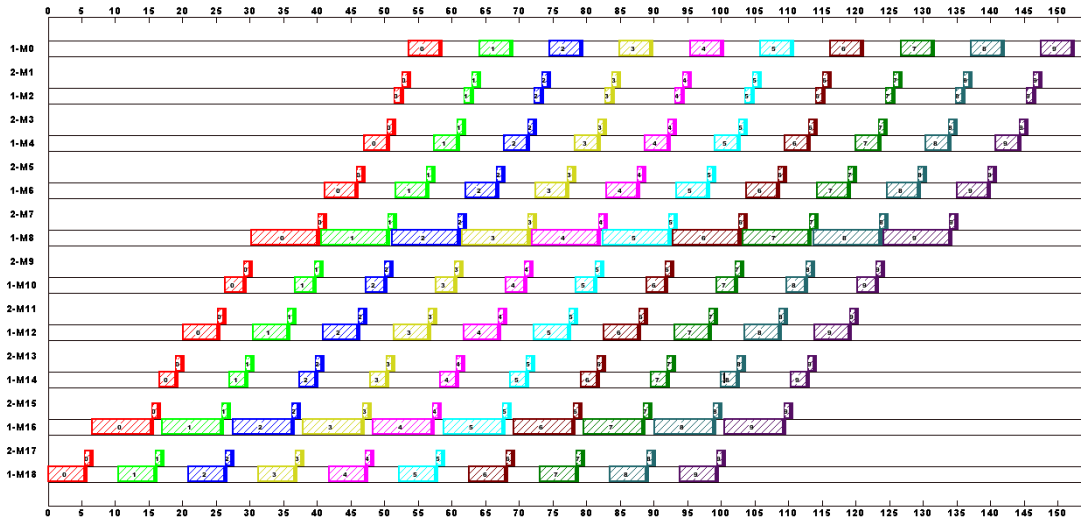

Figure A8-3: The Gantt chart for a NWPMFSS case

Case 4: NWBPMFSS

In Case 4, the data about train directions and train priorities are updated in Table A8-4. In this case, the priorities of some trains are *blocking* and the priorities of the other trains are *no-wait*. In addition, the directions of all trains are *outbound*. Thus,

the problem type is changed to be NWBPMFSS. After applying the SE algorithm, the feasible NWBPMFSS solution is shown in Figure A8-4.

Table A8-4: The definition of train directions and train priorities for a NWBPMFSS case

	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
Train Direction	outbound	outbound	outbound	outbound	outbound	outbound	outbound	outbound	outbound	outbound
Train Priority	no-wait	blocking	no-wait	blocking	no-wait	blocking	no-wait	no-wait	blocking	blocking

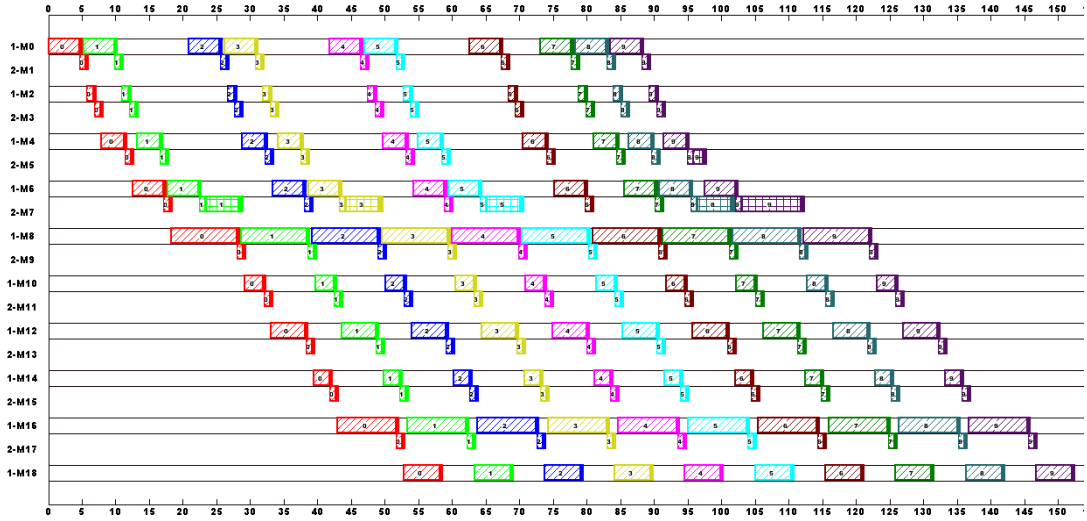

Figure A8-4: The Gantt chart for a NWBPMFSS case

Case 5: NWPMJSS

In Case 5, the data about train directions and train priorities are defined in Table A8-5. In this case, the directions of some trains are *outbound* and the directions of the other trains are *inbound*. In addition, the priorities of all trains are *no-wait*. Thus, the problem type is changed to be NWPMJSS. After applying the SE algorithm, the feasible NWPMJSS solution is shown in Figure A8-5.

Table A8-5: The definition of train directions and train priorities for a NWPMJSS

	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
Train Direction	outbound	inbound	outbound	inbound	outbound	inbound	outbound	inbound	outbound	inbound
Train Priority	no-wait	no-wait	no-wait	no-wait	no-wait	no-wait	no-wait	no-wait	no-wait	no-wait

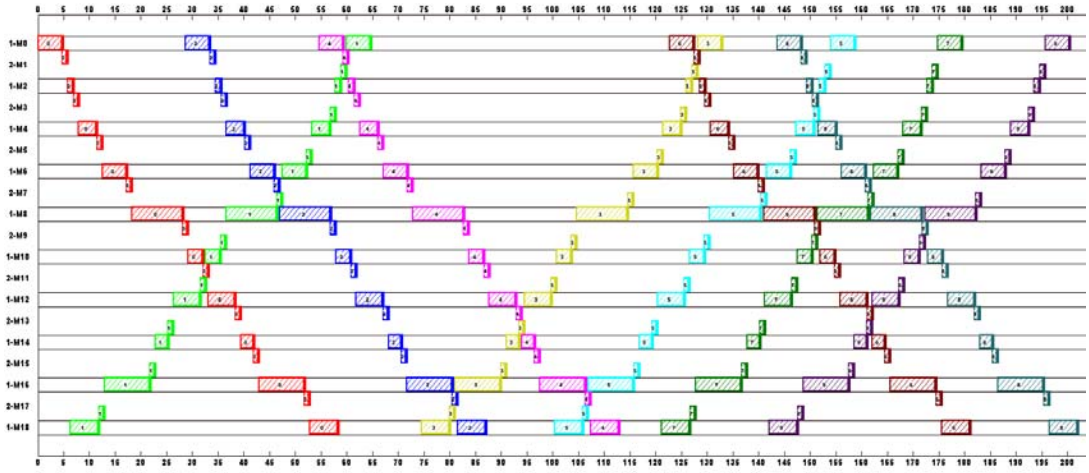

Figure A8-5: The Gantt chart for a NWPMJSS case

Case 6: NWBPMJSS

In Case 6, the data about train directions and train priorities are given in Table A8-6. In this case, the directions of some trains are *outbound* and the directions of the other trains are *inbound*. On the other hand, the priorities of some trains are *blocking* and the priorities of the other trains are *no-wait*. Thus, the problem type is changed to be NWBPMJSS, which is the most complicated and generalised case. After applying the SE algorithm, the feasible NWBPMJSS solution is shown in Figure A8-6.

Table A8-6: The definition of train directions and train priorities for a NWBPMJSS case

	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
Train Direction	inbound	inbound	inbound	inbound	inbound	outbound	outbound	outbound	outbound	outbound
Train Priority	no-wait	blocking	no-wait	blocking	no-wait	blocking	no-wait	blocking	no-wait	blocking

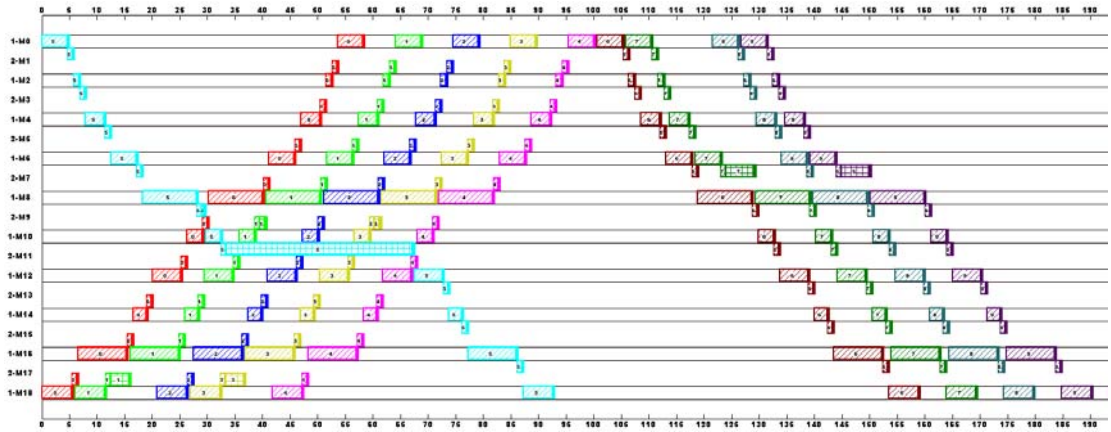

Figure A8-6: The Gantt chart for a NWBPMJSS case

Applying the *SE-BIH* algorithm

We execute more tests by applying the proposed *SE-BIH* algorithm to the above six cases. For comparison, the summary of the results obtained by the *SE* and *SE-BIH* algorithms is presented in Table A8-7. As the *SE* algorithm is a constructive algorithm, for this 10-train 19-section train scheduling example, it is able to quickly build the feasible train timetables with the CPU running time of less than 0.1 second. Thus, it is not necessary to specify the running times of the *SE* algorithm for different cases.

Table A8-7: The summary of the results obtained by the *SE and SE-BIH* algorithms

Cases	Problem Type	Algorithms			
		SE	*SE-BIH*		
		Makespan	The Near-Optimal Order of Trains found	Makespan	CPU Time
Case 1	BPMFSS	152.75	$\{T_0, T_1, T_2, T_3, T_4, T_5, T_7, T_6, T_8, T_9\}$	152.75	16.28
Case 2	BPMJSS	155.34	**$\{T_5, T_6, T_3, T_7, T_0, T_9, T_2, T_8, T_1, T_4\}$**	**140.78**	22.61
Case 3	NWPMFSS	152.75	$\{T_0, T_1, T_2, T_3, T_4, T_5, T_7, T_6, T_8, T_9\}$	152.75	30.45
Case 4	NWBPMFSS	152.75	$\{T_0, T_1, T_2, T_3, T_4, T_5, T_7, T_6, T_8, T_9\}$	152.75	31.32
Case 5	NWPMJSS	202.44	$\{T_0, T_1, T_2, T_3, T_4, T_5, T_7, T_6, T_8, T_9\}$	202.44	34.39
Case 6	NWBPMJSS	190.77	**$\{T_0, T_1, T_3, T_4, T_5, T_7, T_2, T_9, T_6, T_8\}$**	**165.29**	26.86

As shown in Table A8-7, the better solutions for the BPMJSS and NWBPMJSS problems can be found by the *SE-BIH* algorithm and the corresponding train schedules are respectively displayed in Figures A8-7 and A8-8.

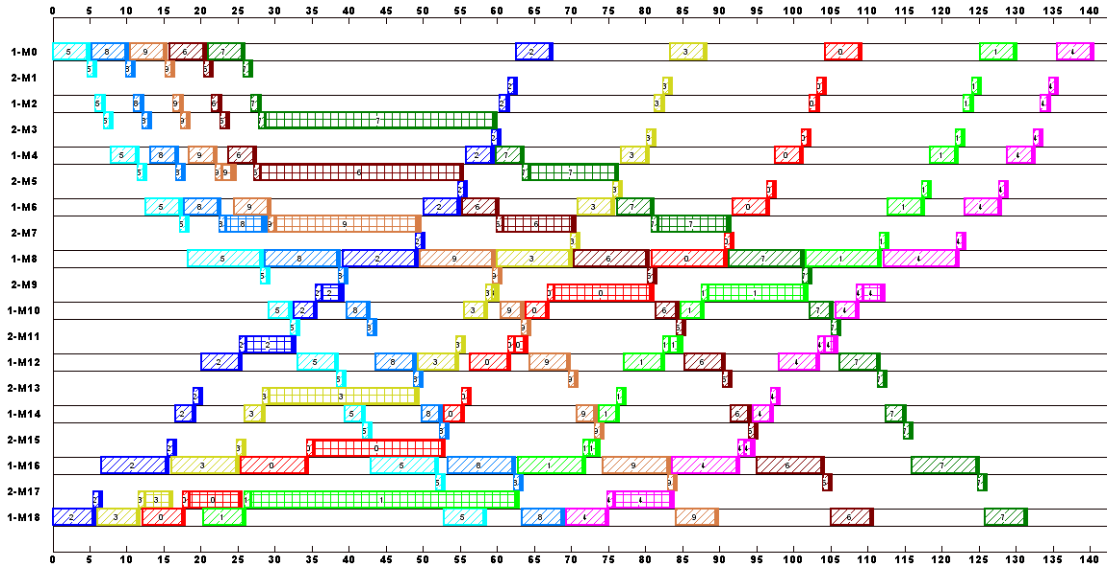

Figure A8-7: The Gantt chart for a near-optimal BPMJSS train schedule obtained by the proposed *SE-BIH* algorithm.

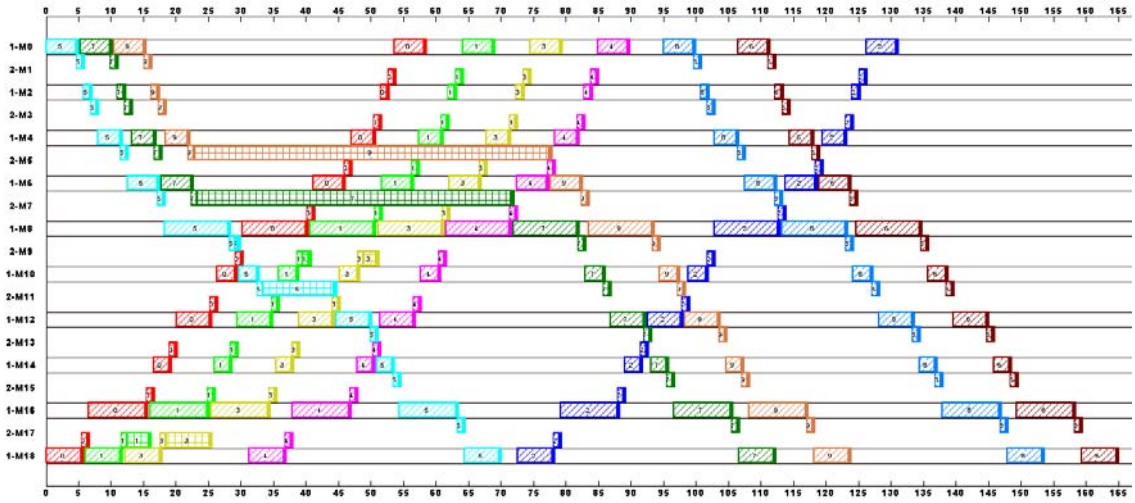

Figure A8-8: The Gantt chart for a near-optimal NWBPMJSS train schedule obtained by the proposed *SE-BIH* algorithm.

For the BPMJSS solution obtained by the *SE-BIH* algorithm, the order of trains found in the *BIH* procedure is $\{T_5, T_6, T_3, T_7, T_0, T_9, T_2, T_8, T_1, T_4\}$. The makespan of this near-optimal train schedule is 140.78, but the running time of the *SE-BIH* algorithm to find this NWBPMJSS schedule is 22.61 seconds due to the fact that more computing efforts are made to guarantee to find the better or optimal solutions. The new order of trains leads to that the makespan decreases from 155.34 in Figure A8-2 to 140.78 in Figure A8-7. In addition, it is observed that the solution quality mainly depend on the sequence of trains in a single-track section (e.g. 1-M8 in Figure A8-7 for this BPMJSS case), which is called a *bottleneck section* because it has the minimum number of gaps (idle times) between trains.

For the NWBPMJSS case, the new order of trains found by the *SE-BIH* algorithm is $\{T_0, T_1, T_3, T_4, T_5, T_7, T_2, T_9, T_6, T_8\}$. The makespan reduces to 165.29 in Figure A8-6 from 190.77 in Figure A8-8. This implies that the percentage of improvement in the efficiency of the overall operation for this transportation task (dispatching 10 trains) in this rail network (20 sections) is up to 13.36%. The more trains are dispatched, the higher percentage of efficiency improvement can be obtained.