SINGLE MACHINE TOTAL WEIGHTED TARDINESS WITH RELEASE DATES

SINGLE MACHINE TOTAL WEIGHTED TARDINESS WITH RELEASE DATES

By

WEI JING, B.Sc.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfilment of the Requirements

for the Degree

Master of Science

McMaster University

MASTER OF SCIENCE (2009) McMaster University
(Computational Engineering & Science) Hamilton, Ontario

TITLE: Single Machine Total Weighted Tardiness With Release Dates

AUTHOR: Wei Jing, B.Sc.

SUPERVISOR: Dr. George Steiner

NUMBER OF PAGES: x, 61.

Abstract

The single machine total weighted tardiness with release dates problem is known to be strongly NP-hard. With a new lower bounding scheme and a new upper bounding scheme, we get an efficient branch and bound algorithm. In the paper, we first introduce the history of the problem and its computational complexity. Second, the lower bounding schemes and the upper bounding schemes are described in detail. We also present all the dominance rules used in the branch and bound algorithm to solve the problem.

In the dominance rules part, we describe the labeling scheme and suggest a data structure for a dominance rule.

Finally, we implement the branch and bound algorithm in C++ for the problem with all the techniques introduced above. We present numerical results produced by the program. Using the same instance generating scheme and the test instances from Dr. Jouglet, our results show that this branch and bound method outperforms the previous approaches specialized for the problem.

Acknowledgments

The thesis was written under the guidance and with the help of my supervisor, Prof. George Steiner. His valuable advices and extended knowledge of the area helped me to do my best while working on the thesis. I am also grateful to Dr. Jouglet for his generous help to my research. My special thanks are to the members of the examination committee: Dr. Christopher Anand, Dr. George Karakostas and Dr. George Steiner.

It would not be possible to complete this thesis without support and help of all members of the School of Computational Engineering and Science. I sincerely thank Laura for her generous assistance through out the development of the project.

Finally, I am indebted to thank my parents and friends for their patience, understanding and continuous support.

Contents

List of Figures

List of Tables

Notations

N: $= \{ 1, ..., n \}$ a set of jobs,

p_j: the processing time of job j,

w_j: the weight of job j,

r_j: the release date of job j,

d_j: the due date of job j,

C_j: the completion time of job j on the machine,

$\sum w_j C_j$: the total weighted completion time for the set of jobs N,

T_j: $= \max(0, C_j - d_j)$ the tardiness of job j,

$\sum w_j T_j$: the total weighted tardiness for the set of jobs N,

σ: partial sequence for the set of jobs N,

$C(\sigma)$: the completion time of the last job in the partial sequence σ,

$C_j(\sigma)$: the completion time of job j in the partial sequence σ,

$TWT(\sigma)$: the total weighted tardiness of the partial sequence σ,

$WT_j(\sigma)$: the weighted tardiness of of job j in the partial sequence σ,

$1||\sum w_iC_i$: the single machine total weighted completion time problem,

$1|r_i|\sum w_iC_i$: the single machine total weighted completion time with release dates problem,

$1||\sum T_i$: the single machine total tardiness problem,

$1|r_i|\sum T_i$: the single machine total tardiness with release dates problem,

$1||\sum w_iT_i$: the single machine total weighted tardiness problem,

$1|r_i|\sum w_iT_i$: the single machine total weighted tardiness with release dates problem,

EDD: Earliest Due Date,

$WSPT$: Weighted Shortest Processing Time,

SPT: Shortest Processing Time,

Chapter 1

Introduction

1.1 Problem Description

Lots of scheduling problems happen everyday in many production companies. Those companies need to assign their limited equipments to different operations in such a way that they can achieve their business goals with the minimum cost. In the scheduling models, equipments are defined as machines and operations are defined as jobs.

The scheduling problem addressed in this paper is the single machine total weighted tardiness with release dates problem, denoted as $1|r_i|\sum w_iT_i$. The problem can be described as follows: there is a set of jobs $N = \{1, ..., n\}$ to be processed on a single machine. The single machine can process only one job at a time and begins to process jobs from time $t = 0$. For any job $i \in N$, it can only be processed after its release date r_i. It needs a positive processing time p_i to be processed uninterruptedly by the machine. When the job is finished at time t on the machine, the completion time C_i of the job is set to t. If the job can not finish before its due date d_i, i.e., $C_i > d_i$, then this will cause a positive weighted tardiness: $w_iT_i = w_i \max(0, C_i - d_i)$. The objective for this problem is to find a job sequence or a job permutation of N with the minimum total weighted tardiness $\sum w_iT_i$.

1.2 Computational Complexity

Different scheduling models have different computational complexities.

Lawler [23] proposes a decomposition theorem for problem $1||\sum T_i$ and points out that the problem can be decomposed to two subproblems with the job which has the largest processing time in N. The two subproblems can use the same decomposition method recursively. Thus the decomposition theorem gives a pseudo-polynomial dynamic programming algorithm and this algorithm takes $O(n^4 \sum_{j=1}^{n} p_j)$ time. Du & Leung [15] show that problem $1||\sum T_i$ is NP-hard in the ordinary sense.

For problem $1||\sum w_i T_i$, Lawler [23] and Lenstra, Rinnooy Kan & Brucker [25] show that this problem is NP-hard in the strong sense.

Because problem $1||\sum w_i T_i$ is a special case of problem $1|r_i|\sum w_i T_i$, problem $1|r_i|\sum w_i T_i$ is NP-Hard in the strong sense too.

1.3 Enumerative Algorithms

This problem has received less research attention. Akturk & Ozdemir [3] propose a branch and bound algorithm combined with some dominance properties for it. Their algorithm can solve test instances with up to 15 jobs. After that, Jouglet, Baptiste & Carlier [21] present a more efficient branch and bound algorithm with new dominance properties. They can solve problems with up to 30 jobs on a personal computer.

Branch and bound algorithm is a powerful enumerative method, which can find the global optimal solutions for many combinatorial optimization problems. This algorithm was proposed originally by Land and Doig [22].

For the scheduling problem, it is better to use a tree structure to present the work flow of a branch and bound algorithm.

The branch and bound algorithm begins from the root node x, in which

no job is sequenced, i.e., the scheduled partial sequence $L_x = \emptyset$ and the unscheduled job set $M_x = N$. Then, for each job $i \in M_x$, we branch to level 1 with a new node y_i and set job i to the first position of the scheduled partial sequence L_{y_i}. If there is no release date or the maximum release date of the unscheduled jobs is smaller than the completion time of the scheduled partial sequence, jobs can be scheduled from right to left in a backward mode. Thus in level 1 of the search tree, there are n branches with n new created nodes. For each new node in level 1 of the tree, $n-1$ branches can be created with a second fixed job. Continuing this branching process, we get a fully expanded search tree with n factorial leaves in level n. The best upper bound UB is updated whenever the search tree expands to level n. It is easy to see that the best upper bound UB equals the global optimum.

In our branch and bound algorithm, the bounding schemes and the dominance rules are employed in the branching process. Every time a new node y of its parent is added into the search tree, we compute a lower bound LB_y for the new node. If the lower bound LB_y is greater than the best known upper bound UB, this new node y is discarded without further branching process. Otherwise, the node is kept in the search tree. We then apply several dominance properties to filter out some jobs in the unscheduled job set $M_y = N - \{i | i \in L_y\}$ and decrease the possible branches from the new node y.

The branch and bound algorithm will stop with an optimal sequence after all the nodes have been eliminated.

1.4 A Simple Example

Suppose we have a 10-job instance as in Table 1.1 for problem $1|r_i|\sum w_i T_i$. If the initial scheduling sequence on the single machine is (1,2,3,4,5,6,7,8,9,10), then the total weighted tardiness for this sequence is 1175. While the optimal

Table 1.1: Data for a 10-job instance

i	1	2	3	4	5	6	7	8	9	10
r_i	4	11	16	9	34	21	31	5	7	33
p_i	6	4	2	10	10	9	9	8	6	5
d_i	39	48	31	30	62	49	40	15	16	44
w_i	4	6	3	10	5	2	9	2	10	7

sequence for this instance is (1,9,4,3,7,10,2,5,8,6) with a minimal total weighted tardiness $\sum w_i T_i = 181$.

1.5 Main Contributions

First, we find a new lower bound method for the problem. For problem $1||\sum w_i T_i$, Potts and Van Wassenhove [29] propose a quickly computed lower bound, which is based on a multiplier adjustment method. We adapt this method to problem $1|r_i|\sum w_i T_i$. We identify decomposed blocks of schedule based on the release dates and apply the multiplier adjustment method [29] to each block by relaxing the release dates of jobs in the block.

The second contribution is a new upper bound method for the problem. For problem $1||\sum w_i T_i$, Congram et al. [12] propose the iterated dynasearch algorithm, which searches in an exponential size neighborhood to find the local optimal solution. We adapt this algorithm to problem $1|r_i|\sum w_i T_i$. Our new upper bound method takes the same time complexity as their algorithm, while the size of the neighborhood is usually smaller than theirs. Because of the release dates, searching for the local optimal solution in the neighborhood would require optimization by two competing criteria: the completion time and the total weighted tardiness of a partial schedule. We present the new dynamic programming heuristic, which finds good quality but not necessarily optimal solution in the neighborhood.

Finally, we suggest a data structure for an important dominance rule. In [21], they use an hash table for the dominance rule. Our choice is a red-black tree, which guarantees that read-only and insert operations take $O(\log n)$ time in the worst case. It improves the performance of the enumerative algorithm for most test instances.

1.6 Outline of the Thesis

In the current chapter, we outline the problem of single machine total weighted tardiness with release dates. In addition, we introduce the computational complexity for the problem. Then we describe current enumerative methods.

Chapter 2 contains the background of the two lower bound schemes in our branch and bound method. The first lower bound scheme is based on a multiplier adjustment method for problem $1||\sum w_i T_i$. The second lower bound uses job splitting technique. Chapter 2 also contains all theoretical results necessary for the implementation.

Chapter 3 is devoted to the different upper bound schemes we used in the branch and bound method. Especially, we describe the new upper bound scheme in detail. The new upper bound scheme is based on a swap neighborhood heuristic method.

In Chapter 4, we present all the dominance rules in the branch and bound method. We provide the necessary background, prove some properties of the dominance rules and suggest a labeling scheme and the data structure.

Chapter 5 describes the generating scheme of the test instances and provides the computational results of the branch and bound algorithm. We give detailed comparison and discussion in this chapter.

Finally, Chapter 6 contains concluding remarks and suggestions for future work.

Chapter 2

Lower Bounding Schemes

In this chapter, we present the details of the two lower bounding schemes in our branch and bound method.

2.1 Lower bound I

The first lower bound is based on the multiplier adjustment method proposed by Potts and Van Wassenhove [29]. We review the method in the following sections and then describe the new lower bounding scheme.

2.1.1 The Lagrangian Problem

We relax problem $1|r_i|\sum w_iT_i$ to $1||\sum w_jT_j$ for the first lower bound calculation. Here we recall how they [29] relax problem $1||\sum w_jT_j$ to a Lagrangian problem first.

Suppose there is a set of jobs $N = \{1, ..., n\}$ to be processed on the single machine. The tardiness of job i is defined as

$$T_i = max(0, C_i - d_i), i = 1, ..., n$$

From the definition of the tardiness T_i, we know that

$$T_i \geq C_i - d_i, i = 1, ..., n$$

Then we can describe the problem $1||\sum w_jT_j$ as (P [29]):

$$\begin{aligned} & \min \quad \sum_{i=1}^{n} w_iT_i \\ (\mathrm{P}) \quad & \text{s.t.} \quad T_i \geq 0, i = 1, ..., n, \\ & \qquad T_i \geq C_i - d_i, i = 1, ..., n, \end{aligned} \tag{2.1.1}$$

where the domain is all the possible permutations of job 1 to job n.

Their Lagrangian relaxation is then based on constraints (2.1.1). Constraints (2.1.1) can be transformed to

$$-T_i + (C_i - d_i) \leq 0, i = 1, ..., n \tag{2.1.2}$$

Suppose $u = (u_1, ..., u_n)$ is a vector and subject to $u_i \geq 0, i = 1, ..., n$. It is easy to see that constraints (2.1.2) can be transformed to

$$-u_iT_i + u_i(C_i - d_i) \leq 0, i = 1, ..., n \tag{2.1.3}$$

By adding constraints (2.1.3) to the objective of problem (P), we get the Lagrangian problem (LR [29]):

$$\begin{aligned} & L(u) = \min \textstyle\sum_{i=1}^{n} [(w_i - u_i)T_i + u_i(C_i - d_i)] \\ (LR) \quad & \\ & \text{s.t. } T_i \geq 0, u_i \geq 0, i = 1, ..., n \end{aligned}$$

Even if we choose the nonnegative vector u arbitrarily, according to Fisher and Geoffrion [16, 17], we know that the Lagrangian problem (LR) gives a lower bound for problem (P). In [29], they point out that it is possible that $L(u) = -\infty$ if there exists some i such that $u_i > w_i$. To avoid this meaningless situation, they restrict the choice of vector u. Besides $0 \leq u_i$ for every $i, i = 1, ..., n$, vector u is chosen subject to $u_i \leq w_i, i = 1, ..., n$. After setting $T_i = 0$ for every $i, i = 1, ..., n$, by the weighted shortest processing time rule [31], they point out that the jobs in weighted shortest processing time order with the new weight u_i and the processing time p_i gives the solution of

the Lagrangian problem (LR). The question is then transferred to how to find the value of vector u to make $L(u)$ as large as possible. In section 2.1.2, we introduce the multiplier adjustment method and give the detailed steps to find the value of vector u.

2.1.2 The Multiplier Adjustment Method

To compute a lower bound for some single machine problems, Fisher [16] uses the subgradient optimization technique to find the multipliers for the Lagrangean problems relaxed from the original problems. The subgradient optimization is a pseudo-polynomial dynamic programming method. Van Wassenhove [33] proposes the multiplier adjustment method to replace the subgradient optimization technique and applies it to some scheduling problems with better results. In [29], a corresponding multiplier adjustment method is proposed for problem $1||\sum w_jT_j$. Compared with the subgradient optimization technique, this multiplier adjustment method is not necessarily optimal, but it can be quickly computed in $O(n \log n)$ time.

To find the value of vector u, an initial sequence by some heuristics is needed. Suppose the initial heuristic sequence is renumbered as $(1, ..., n)$ and the completion time for job i in the sequence is C_i^*. From section 2.1.1, we know that vector u is chosen subject to $u_i \leq w_i, i = 1, ..., n$ and the weighted shortest processing time rule, i.e., $u_i/p_i \geq u_{i+1}/p_{i+1}, i = 1, ..., n-1$. To make $L(u)$ as large as possible, the Lagrangian problem (LR) can be transformed to ($LR(C^*)$ [29]):

$$\text{LR(C*)} \quad \begin{aligned} \max \quad & \sum_{i=1}^{n} u_i(C_i^* - d_i) \\ \text{s.t.} \quad & u_i/p_i \geq u_{i+1}/p_{i+1}, i = 1, ..., n-1 \qquad (2.1.4) \\ & 0 \leq u_i \leq w_i, i = 1, ..., n \qquad (2.1.5) \end{aligned}$$

By Lemma 2.1.1, they show that constraints (2.1.5) can be transformed to [29]:

$$0 \leq u_i \leq \bar{w}_i, i = 1, ..., n \tag{2.1.6}$$

where $\bar{w}_i = p_i \min_{h \in \{1,...,i\}} \{w_h/p_h\}$.

Lemma 2.1.1 [29]

The solution of problem $LR(C^*)$ stays same with constraints (2.1.6).

To find the solution of problem $LR(C^*)$, we first need a procedure to get set $V = \{v_1, ..., v_r\}$, where $v_1, ..., v_r \in N$ and can be computed as follows [29].

Procedure SV. [29]

Step 1. Suppose the initial heuristic sequence is renumbered as $(1, ..., n)$. Let $S_j = \sum_{i=1}^{j} p_i(C_i^* - d_i)$, for $j = 1, ..., n$. /*— S is an array —*/

Step 2. Let $v = 0, k = 1, S_0 = 0$ and $V = \emptyset$. /*— here we set $v_0 = 0, S_0 = 0$ —*/

Step 3. If $S_k > S_v$, then set $V = V \cup \{k\}$ and $v = k$; else set $k = k + 1$ and goto Step 5. /*— Based on the set $V = \{v_1, ..., v_{k-1}\}$ we have found, we select the smallest v_k, which needs to satisfy: $v_k > v_{k-1}$ and $S_{v_k} > S_{v_{k-1}}$. —*/

Step 4. Set $k = k + 1$.

Step 5. If $k \leq n$, then goto Step 3.

Step 6. Return set V, stop.

The following theorem shows how we use set V to solve problem $LR(C^*)$.

Theorem 2.1.2 [29]

The solution of problem $LR(C^*)$ is given by $u_i = u_i^*$ for $i = 1, ..., n$ where

$$u_i^* = u_{i+1}^*(p_i/p_{i+1}) \qquad i \notin V, i \neq n \tag{2.1.7}$$

$$u_i^* = \bar{w}_i \qquad i \in V \tag{2.1.8}$$

$$u_n^* = 0 \qquad if\ n \notin V. \tag{2.1.9}$$

In Theorem 2.1.2, if the initial heuristic sequence is given by weighted shortest processing time order with the original weight w_i and the processing time p_i, then $\bar{w}_i = w_i, i = 1, ..., n$.

2.1.3 Implementation of Multiplier Adjustment Method

The multiplier adjustment method computes u_i^* for $i = 1, ..., n$, which satisfies (2.1.7)-(2.1.9). The input of the method is the unscheduled job set N. Instead of sets, we use four arrays to store the necessary data. Following is the detailed computing procedure.

Procedure MAM.

Step 1. Sort all jobs of N in weighted shortest processing time order and renumber the sequence as $(1, ..., n)$.

Step 2. Set $C_0 = 0, V_0 = 0, S_0 = 0$ and $W_0 = 0$. Let $i = 1$ and $k = i - 1$. /*— C, V, S, W are arrays —*/

Step 3. If $i > n$, then goto Step 7. /*— from Step 3 to Step 7, Procedure SV. —*/

Step 4. Set $V_i = 0, C_i = C_{i-1} + p_i$ and $S_i = S_{i-1} + p_i(C_i - d_i)$.

Step 5. If $S_i > S_k$, then set $V_i = i$ and $k = i$.

Step 6. Let $W_i = w_i$ and $i = i + 1$, goto Step 3. /*— because the initial order is WSPT, $\bar{w}_i = w_i$ —*/

Step 7. Let $j = n$ and $LB = 0$. If $V_j = 0$, then set $W_j = 0$; else let $LB = W_j(C_j - d_j)$.

Step 8. Let $j = j - 1$. If $j \leq 0$, then goto Step 11. /*— from Step 8 to Step 11, compute u_i^* and LB —*/

Step 9. If $V_j = 0$, set $W_j = W_{j+1}(p_j/p_{j+1})$.

Step 10. Set $LB = LB + W_j(C_j - d_j)$ and goto Step 8.

Step 11. return LB, stop.

It is clear that the multiplier adjustment method contains one sorting step and two loops. The computational complexity of this method is $O(n \log n)$.

2.1.4 The Heuristic Decomposition Algorithm

In this section we propose the new lower bounding scheme. Suppose the unscheduled job set is N. The basic idea is: first, we sort all the unscheduled jobs in nondecreasing release date order and renumber them as $(1, ..., n)$. The jobs are decomposed into different blocks. Then we compute the total weighted tardiness or a lower bound of it for each block. In the last step, by adding the contribution of each block together, we get the final lower bound for the whole job set N. The idea of decomposition of jobs into blocks is not new, see [19].

By Figure 2.1, it is clear that this decomposition is based on the distribution of the release dates.

Here we give the detailed steps of the decomposition algorithm.

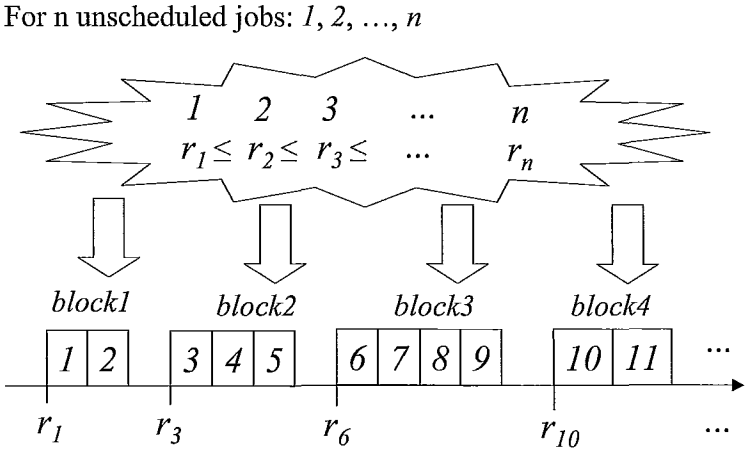

Figure 2.1: Idea of the heuristic decomposition algorithm.

Heuristic Decomposition Algorithm.

Step 1. Suppose the unscheduled job set is N. Sort all jobs of N in nondecreasing release date order, renumber the sequence as $(1, ..., n)$ and let $S = \{1, ..., n\}$.

Step 2. Set $\Delta = r_1$, start the first block B_1 and set $B_1 = \emptyset$.

Step 3. If $S \neq \emptyset$, suppose the current first job is $j, j \in S$ and current block is B_p, set $B_p = B_p \cup \{j\}$; else goto Step 7.

Step 4. Set $\Delta = \Delta + p_j$ and $S = S - \{j\}$.

Step 5. If $S = \emptyset$, then goto Step 7.

Step 6. If $r_{j+1} \geq \Delta$, start a new block B_{p+1}, set $B_{p+1} = \emptyset$ and $\Delta = r_{j+1}$. Goto Step 3;

Step 7. Compute the lower bound LB_p of each block B_p.

Step 8. Let the final lower bound $LB = \sum LB_p$, return LB.

It is clear that the final lower bound generated by the heuristic decomposition algorithm gives a lower bound for problem $1|r_i|\sum w_iT_i$.

The question is how we get the total weighted tardiness or the lower bound for each block. According to the survey by Abdul-Razaq, Potts & Van Wassenhove [1], the fastest algorithm for problem $1||\sum w_jT_j$ uses a quickly computed lower bound method, which is based on a multiplier adjustment method and takes $O(n\log n)$ time, see [29]. For Lower Bound I, we choose this method to compute LB_p. In the previous sections, we have given a detailed introduction to the multiplier adjustment method proposed by Potts & Van Wassenhove [29].

2.1.5 Computational Complexity of Lower Bound I

Section 2.1.4 gives the detailed steps of the heuristic decomposition algorithm. We have eight steps for the algorithm. The sort operation in Step 1 takes $O(n\log n)$ time. Step 2 takes $O(1)$ time. From Step 3 to Step 6, it will cost $O(n)$ time to find all blocks. Then in Step 7, for each block, we apply the $O(n\log n)$ multiplier adjustment method by Potts & Van Wassenhove [29]. It is easy to see that the one block situation takes the largest time $O(n\log n)$. Step 8 takes no more than $O(n)$ time. Finally, the time complexity of the Heuristic Decomposition Algorithm is $O(n\log n)$.

2.2 Lower Bound II

Similarly to [2], we get the second lower bound from the single machine total weighted completion time problem with release dates. This lower bound is based on the job splitting technique proposed by Belouadah, Posner & Potts [5].

2.2.1 Job Splitting

In [24], Lawler uses the job splitting technique to minimize total weighted completion time with precedence constraints. The basic idea of job splitting can be described as follows: suppose for any given sequence σ of N, there is a job $i \in \sigma$, and if we split job i to two new jobs i_1 and i_2 subject to $p_i = p_{i_1} + p_{i_2}, w_i = w_{i_1} + w_{i_2}$, then we get a new sequence σ'. In the new sequence σ', job i_1 is required to be scheduled before job i_2 and there is no other job scheduled between them. Suppose the weighted completion time of σ is $WC(\sigma)$. Compared with the old sequence σ, the weighted completion time of σ' is $WC(\sigma') = WC(\sigma) - p_{i_2} w_{i_1}$.

For the situation of splitting one job into more pieces, we first recall some definitions from [5].

Definition 2.2.1 [5]

Let P denote the original total weighted completion time problem without job splitting. Let P_1 be the identical problem to P, except that job i is split into k pieces $i_1, ..., i_k$, where $p_{i_1} + ... + p_{i_k} = p_i$ and $w_{i_1} + ... + w_{i_k} = w_i$. In problem P_1, these k new jobs are required to be contiguously sequenced in the order $(i_1, ..., i_k)$. Let $N_1 = \{1, ..., i-1, i_1, ..., i_k, i+1, ...n\}$ denote the job set of problem P_1.

In Definition 2.2.1, if the k pieces $i_1, ..., i_k$ of job i are subject to

$$w_{i_1}/p_{i_1} = ... = w_{i_k}/p_{i_k} = w_i/p_i,$$

then this is a *simple split* [5].

If the k pieces $i_1, ..., i_k$ of job i are not required to be contiguously sequenced, problem P_1 is further relaxed to P_2:

Definition 2.2.2 [5]

Let problem P_2 be the identical problem to P_1, except that the new jobs $i_1, ..., i_k$ are not required to be contiguously sequenced. But the k pieces $i_1, ..., i_k$

are required to be scheduled in consistent order, i.e., job i_{j-1} must be sequenced before job $i_j (j = 2, ..., k)$.

If there is no limitation on the weight of the k pieces $i_1, ..., i_k$ of job i, problem P_2 can be further relaxed to $P_2{}'$:

Definition 2.2.3 [5]

Let problem $P_2{}'$ denote the same problem as problem P_2, except that the weight of k pieces $i_1, ..., i_k$ defined as $w_{i_1}{}', ..., w_{i_k}{}'$ need to satisfy only $w_{i_1}{}' + ... + w_{i_k}{}' = w_i$.

Using the *2 job split* iteratively, we can achieve the *k job split*. It is easy to see the relation of the total weighted completion time between problem P and P_1:

Theorem 2.2.4 [27, 5]

For any given feasible sequence σ of problem P, if σ_1 is the corresponding sequence for problem P_1, then

$$\sum_{i \in N} w_i C_i(\sigma) - \sum_{i \in N_1} w_i C_i(\sigma_1) = CBRK, \quad (2.2.1)$$

where

$$CBRK = \sum_{h=1}^{k-1} w_{i_h} \sum_{j=h+1}^{k} p_{i_j}.$$

By Definition 2.2.2, Theorem 2.2.5 shows that the optimal solution of problem P_2 gives a lower bound for the original problem P.

Theorem 2.2.5 [5]

If $\sigma_2{}^*$ is an optimal sequence for P_2, then

$$\sum_{i \in N_1} w_i C_i(\sigma_2{}^*) + CBRK \leq \sum_{i \in N_1} w_i C_i(\sigma_1{}^*) + CBRK = \sum_{i \in N} w_i C_i(\sigma^*),$$

where σ^* is an optimal sequence for problem P and $\sigma_1{}^*$ is the corresponding optimal sequence for P_1.

To get a better lower bound, based on Theorem 2.2.5, Belouadah et al. propose the following result.

Theorem 2.2.6 [5]

Suppose $\sigma_2{}^*$ is an optimal sequence for problem P_2, $\alpha_2{}^*$ is an optimal sequence for problem $P_2{}'$ and CBRK' is obtained from (2.2.1) by using $w_{i_h}{}'$ instead of w_{i_h}. If $\sum_{h=1}^{j} w_{i_h}{}' \leq \sum_{h=1}^{j} w_{i_h}$ for j=1,...,k, then

$$\sum_{i\in N_1} w_i{}' C_i(\alpha_2{}^*) + CBRK' \geq \sum_{i\in N_1} w_i C_i(\sigma_2{}^*) + CBRK.$$

2.2.2 Lower Bound From General Split

In Definition 2.2.1, if the k pieces $i_1, ..., i_k$ of job i are not subject to

$$w_{i_1}/p_{i_1} = ... = w_{i_k}/p_{i_k} = w_i/p_i,$$

then this is a *general split* [5].

In [5], the authors present a greedy heuristic for problems with parallel chain precedence constraints. Then based on the heuristic, they give the procedure for finding good *general splits.*

Heuristic H. [5]

Step 1. Let $S = \{1, ..., n\}$. For each job $i \in S$, let B_i be the set of predecessors and A_i be the set of successors. Let $u = 0, t = 0$ and $WC = 0$.

Step 2. Let $S' = \{i | i \in S, B_i \cap S = \emptyset\}$. If $t < \min_{i\in S'}\{r_i\}$, then set $t = \min_{i\in S'}\{r_i\}$.

Step 3. Find job $i \in S'$ such that $w_i/p_i = \max_{l\in S''}\{w_l/p_l\}, S'' = \{l | l \in S', r_l \leq t\}$. Let $u = u+1, \sigma^H(u) = i, t = t+p_i, C_i{}^H = t, WC = WC+w_i t$ and $S = S - \{i\}$. If $S \neq \emptyset$, goto Step 2; else stop.

The following result shows under what condition, the *Heuristic H* can give an optimal sequence for problems with *parallel chain precedence constraints* - precedence constraints which consist of several chains of otherwise unrelated jobs.

Theorem 2.2.7 [5]

Heuristic H generates an optimal sequence for a problem with parallel chain precedence constraints if the following condition is satisfied throughout the execution of Heuristic H: for each $j \notin A_i$ *with* $r_j < C_i^H < C_j^H$ *and* $k \in A_j \cup \{j\}$, *we have* $p_i/w_i \leq p_k/w_k$.

The conditions state that if a job i is processed before an available job j, then job i must have a 'better' p/w ratio than job j or any of its successors. The job splitting in the following procedure always ensures that all pieces satisfy the conditions of Theorem 2.2.7.

Procedure SP. [5]

Step 1. For each job $i \in N$, *let* $r_{i_1} = r_i, p_{i_1} = p_i$ *and* $w_{i_1} = w_i$. *Let* $S = \{1_1, ..., n_1\}, u = 0, t = 0, WC = 0$ *and* $CBRK = 0$.

Step 2. If $t < \min_{i \in S}\{r_i\}$, *then let* $t = \min_{i \in S}\{r_i\}$. *Find job* $i \in S$ *such that* $w_i/p_i = \max_{l \in S'}\{w_l/p_l\}, S' = \{l | l \in S, r_l \leq t\}$, *and set* $i_k = i$. *If there exists a job* $j \in S$ *such that* $r_j < t + p_{i_k}$ *and* $w_j/p_j > w_{i_k}/p_{i_k}$, *goto Step 4.*

Step 3. Let $u = u + 1, \sigma(u) = i_k, t = t + p_{i_k}, C_{\sigma(u)} = t, WC = WC + w_{i_k}t$ *and* $S = S - \{i_k\}$. *If* $S = \emptyset$, *goto Step 7; else goto Step 2.*

Step 4. Find job $j \in S$ *such that* $r_j = \min_{l \in S'}\{r_l\}, S' = \{l | l \in S, w_l/p_l > w_{i_k}/p_{i_k}\}$. *Split* i_k *into two new jobs* i' *and* i''. *Set* $r_{i'} = r_{i''} = r_i, p_{i'} = r_j - t, p_{i''} = p_{i_k} - p_{i'}$.

Step 5. If this is a simple split, then set $w_{i'} = p_{i'}w_i/p_i$ *and goto Step 6; else let* $E_1 = \{h | h \in S - \{i_k\}, r_h < r_j\}$ *and* $E_2 = \{\sigma(h) | h \leq u, \sigma(h) \neq i_j$ *for* $j = 1, ..., k-1, C_{\sigma(h)} > r_i, p_{\sigma(h)}/w_{\sigma(h)} > p_{i''}/w_{i_k}\}$. *If* $E_1 = \emptyset$, *set* $\rho_1 = +\infty$; *else set* $\rho_1 = \min_{h \in E_1}\{p_h/w_h\}$. *If* $E_2 = \emptyset$, *set* $\rho_2 = p_{i''}/w_{i_k}$; *else set* $\rho_2 = \max_{h \in E_2}\{p_h/w_h\}$. *Let* $w_{i'} = \max\{p_{i'}/\rho_1, w_{i_k} - p_{i''}/\rho_2\}$.

Step 6. Let $w_{i''} = w_{i_k} - w_{i'}$ *and* $S = S \cup \{i''\} - \{i_k\}$. *Rename* i' *to* i_k *and* i'' *to* i_{k+1}. *Set* $u = u + 1, \sigma(u) = i_k, t = r_j, C_{\sigma(u)} = t, WC = WC + w_{i_k}t, CBRK = CBRK + w_{i_k}p_{i_{k+1}}$, *goto Step 2.*

Step 7. Compute $LB_{SP} = WC + CBRK$ *and stop.*

By Theorem 2.2.8, they show that *Procedure SP* provides a better lower bound if the split in step 5 is a general split.

Theorem 2.2.8 [5]

If LB_{SS} *denote the lower bound generated by Procedure SP with simple splits and* LB_{GS} *denote the lower bound generated by Procedure SP with general splits, then* $LB_{GS} \geq LB_{SS}$.

The computational complexity of *Procedure SP* is given by the following result.

Theorem 2.2.9 [5]

Procedure SP takes $O(n \log n)$ *time with simple splits. Procedure SP takes* $O(n^2)$ *time with general splits.*

From *Procedure SP*, we get a lower bound for problem $1|r_i|\sum w_iT_i$.

Corollary 2.2.10 [2]

If LB_{SP} *is generated by Procedure SP, then* $LB_{SP} - \sum_{i=1}^{n} w_i d_i$ *provides a lower bound for problem* $1|r_i|\sum w_iT_i$.

2.2.3 Implementation of Lower Bound II

Suppose the input of lower bound II is the set of unscheduled jobs N. In our program, we use three arrays S, C and T to store necessary data to compute the lower bound. In this implementation, only general splits are used. We get E_1 from S and E_2 from T.

Following is the detailed computing procedure of lower bound II. A dummy job 0 is needed. For job 0, we have $r_0 = w_0 = p_0 = d_0 = 0$.

Implementation of Lower Bound II.

Step 0. Sort all jobs of N in nondecreasing release date order, renumber the sequence as $(1, ..., n)$ and let $S = \{1, ..., n\}$.

Step 1. Let $C_0 = 0$ and T_0 be the dummy job 0. Set $t = 0, WC = 0, sn = n, tn = 0, WD = \sum_{i=1}^{n} w_i d_i$ and $CBRK = 0$. /*— C, T, S are arrays; C and T are used to compute E_2; $T = \{\sigma(h) | h \leq u, \sigma(h) \neq i_j, j = 1, ..., k-1\}$ and $C = \{C_{\sigma(h)} | h \leq u, \sigma(h) \neq i_j, j = 1, ..., k-1\}$; sn is an index of $S's$ last member; tn is an index of $T's$ last member —*/

Step 2. If $sn = 0$, goto Step 8. /*— if S is empty, return the final result and stop —*/

Step 3. Let $i = S_1$. For each $m \in S$, if $r_m < r_i$, then set $i = m$. If $t < r_i$, set $t = r_i$. For each $m \in S$, if $r_m \leq t$ and $w_m/p_m > w_i/p_i$, then set $i = m$. Let $i_k = i$ and $j = 0$. For each $m \in S$, if $r_m < t + p_{i_k}$ and $w_m/p_m > w_{i_k}/p_{i_k}$, then let $j = m$ and goto Step 5. /*— if S is not empty, find job i (S_i) and job j (S_j); same as Step 2 of Proc. SP —*/

Step 4. Set $t = t + p_{i_k}, WC = WC + w_{i_k}, tn = tn + 1, T_{tn} = S_{i_k}$ and $C_{tn} = t$. Let $S_{i_k} = S_{sn}$ and $sn = sn - 1$. Goto Step 2. /*— if job j is not found, then job i is processed without split; update S, C, T, WC, sn and tn; $S_{i_k} = S_{sn}, sn = sn - 1$ means $S = S - \{S_{i_k}\}$; $tn = tn + 1, T_{tn} = S_{i_k}$ means $T = T \cup \{S_{i_k}\}$; same as Step 3 of Proc. SP —*/

Step 5. For each $m \in S$, if $r_m < r_j$ and $w_m/p_m > w_{i_k}/p_{i_k}$, then set $j = m$. Let $p' = r_j - t, p'' = p_{i_k} - p'$. /*— find job j with the max w/p and split i_k to i' and i''; same as Step 4 of Proc. SP —*/

Step 6. Let $E_1 = \emptyset$. For each $m \in S$, if $m \neq i_k$ and $r_m < r_j$, then set $E_1 = E_1 \cup \{m\}$. Let $E_2 = \emptyset$. For each $m \in T$, if $C_m > r_{i_k}$ and $p_m/w_m > p''/w_{i_k}$, then set $E_2 = E_2 \cup \{m\}$. If $|E_1| = 0$, set $\rho_1 = +\infty$; else let $\rho_1 = \min_{h \in E_1}\{p_h/w_h\}$. If $|E_2| = 0$, set $\rho_2 = p''/w_{i_k}$; else let $\rho_2 = \max_{h \in E_2}\{p_h/w_h\}$. Let $w' = \max\{p'/\rho_1, w_{i_k} - p''/\rho_2\}$. /*— compute $E_1, E_2, w_{i'}$ and $w_{i''}$; same as Step 5 of Proc. SP —*/

Step 7. Let $w'' = w_{i_k} - w', t = r_j, WC = WC + w' * t, CBRK = CBRK + w' * p'', w_{i_k} = w''$ and $p_{i_k} = p''$, goto Step 2. /*— update S, WC and $CBRK$; $w_{i_k} = w'', p_{i_k} = p''$ means $S = S \cup \{i''\} - \{i_k\}$; same as Step 6 of Proc. SP —*/

Step 8. Let $LB_{SP} = WC + CBRK$ and return $LB_{SP} - WD$.

Chapter 3

Upper Bounding Schemes

In this chapter, we present the details of the two upper bounding schemes in our branch and bound method. The first upper bound scheme is based on iterated dynasearch [12], a fast swap neighborhood heuristic method. The second one is a dispatching rule [34].

3.1 Upper Bound I

The iterated dynasearch is proposed by Congram et al. [12] for problem $1||\sum w_jT_j$. This heuristic is a large neighborhood local search algorithm. The iterated dynasearch is based on the idea of dynasearch by Potts et al. [28]. Using dynamic programming techniques, the iterated dynasearch algorithm takes $O(n^3)$ time to search in an exponential size neighborhood and find the local optimal sequence for problem $1||\sum w_jT_j$. We propose a modified version of this heuristic algorithm to get good upper bounds for problem $1|r_i|\sum w_iT_i$.

3.1.1 Swap and Independent Dynasearch Swaps

The domain for problem $1||\sum w_jT_j$ is all the possible permutations of job set $N = \{1, ..., n\}$. It is clear that the size of the solution space for a branch and bound algorithm is $n!$.

Different from the branch and bound algorithm, an iterative improve-

ment local search algorithm searches for a near-optimal solution in a smaller *neighborhood*. The main idea of these local search algorithms is to get a better sequence by exchanging jobs in the initial given sequence. If a better sequence is found, then the algorithm starts a new search based on the new sequence. The algorithm will repeat the search until any job exchanging can not generate a better sequence.

In [12], *k-exchange neighborhood* refers to all sequences that can be generated by exchanging k jobs from a given sequence.

Based on the definition of *k-exchange neighborhood*, we can define what is a *swap*.

Definition 3.1.1 [12]

For a given sequence σ of N, swap is a 2-exchange neighborhood and the size of it is $n(n-1)/2$.

In each iteration, a simple iterative improvement local search algorithm searches for a better sequence in the *2-exchange neighborhood* of the current sequence. There are two types of search strategies: *first-improve* and *best-improve* [12]. The *first-improve* local search algorithm searches for the first better *swap* in each iteration, while the latter searches for the best *swap*.

Different from the simple iterative improvement local search algorithms, dynasearch performs a series of swaps in the given sequence. The swaps must be *independent* [12]. The search strategy of dynasearch is $best - improve$. Here we recall the definition of *independent dynasearch swap*.

Definition 3.1.2 [12]

For a given sequence σ of N, we choose m pairs of jobs to perform the series of swaps. If for any two swaps (i,j) and (k,p) we have $\max(i,j) < \min(k,p)$ or $\min(i,j) > \max(k,p)$, then the swaps are called independent dynasearch swaps.

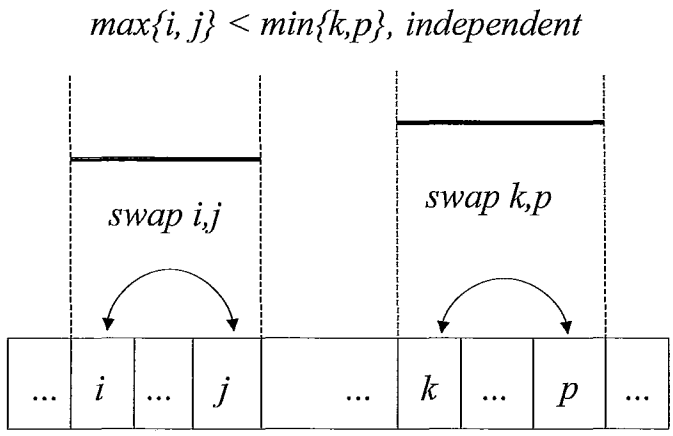

Figure 3.1: Independent dynasearch swaps.

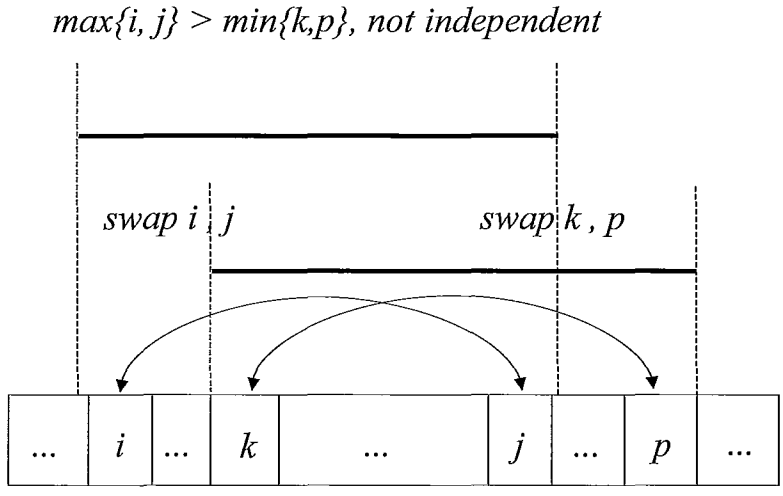

Figure 3.2: These swaps are not independent.

Figure 3.1 and Figure 3.2 show whether the swaps are independent.

The neighborhood size of the *independent dynasearch swaps* is given by the following result.

Lemma 3.1.3 [12]

The neighborhood size of the independent dynasearch swaps is $2^{n-1} - 1$.

3.1.2 Dynasearch for $1||\sum w_j T_j$

In this section, we recall how they [12] construct the dynamic programming algorithm to find the independent dynasearch swaps for a given sequence.

Suppose the initial given sequence is $\sigma = (\sigma(1), ..., \sigma(n))$. The new sequence σ' is built by the dynamic programming algorithm in a forward enumeration mode. In this algorithm, we build the new sequence from the 1 job partial sequence to the n-job sequence. In each step, a new job is added after the last job of the current partial sequence. Then the dynamic programming algorithm enters into a new state and searches for the best independent dynasearch swaps of the new partial sequence. Considering the situation when no job is selected, it is easy to see that we have $n + 1$ states in the algorithm. In [12], after searching for the best independent swaps with the partial sequence $(\sigma(1), ..., \sigma(k))$, where $k \in \{0, ..., n\}$, they define the new sequence to be in state (k, σ), for $k = 0, ..., n$. The sequence in state (n, σ) with the best independent dynasearch swaps is the new sequence σ' we want.

Suppose for $i = 0, ..., k-1$, where $k \in \{0, ..., n\}$, we have found and stored some information of state (i, σ), such as the sequence σ_i and the total weighted tardiness $TWT(\sigma_i)$, where σ_i is a permutation of the partial sequence $(\sigma(1), ..., \sigma(i))$ derived through the best independent dynasearch swaps for these jobs. Then for state (k, σ), finding σ_k must rely on the information of previous states from $(0, \sigma)$ to $(k-1, \sigma)$. Following is the detailed procedure to find σ_k and $TWT(\sigma_k)$.

Procedure SK. [12]

Step 1. Suppose we have found and stored σ_i and $TWT(\sigma_i)$ for $i = 0, ..., k-1$, for some $k \in \{0, ..., n\}$.

Step 2. Let $j = k - 1$. Job $\sigma(k)$ is directly added after σ_j and no need to swap job $\sigma(k)$ with other jobs in σ_j. Let $\sigma_k = (\sigma_j, \sigma(k))$ and compute the total weighted tardiness $TWT(\sigma_k)$.

Step 3. Let $j = j - 1$.

Step 4. If $j \geq 0$, construct the sequence $\sigma'_k = (\sigma_j, \sigma(j+1), \sigma(j+2), ..., \sigma(k-2), \sigma(k))$, then swap job $\sigma(k)$ and job $\sigma(j+1)$ in sequence σ'_k; else goto Step 7.

Step 6. If $TWT(\sigma'_k) < TWT(\sigma_k)$, let $\sigma_k = \sigma'_k$. Goto Step 3.

Step 7. Return σ_k and $TWT(\sigma_k)$. Stop.

Their dynamic programming algorithm can be described as one recursion function with two initial conditions($G(\sigma_k)$ [12]):

$$G(\sigma_k) = \begin{cases} 0, k = 0 \\ w_{\sigma(1)} \max(0, p_{\sigma(1)} - d_{\sigma(1)}), k = 1 \\ \min \begin{cases} G(\sigma_{k-1}) + w_{\sigma(k)} \max(0, \sum_{q=1}^{k-1} p_{\sigma(q)} - d_{\sigma(k)}), \\ \min_{0 \leq i < k-1} \{ \\ \quad G(\sigma_i) + w_{\sigma(k)} \max(0, \sum_{q=1}^{i} p_{\sigma(q)} + p_{\sigma(k)} - d_{\sigma(k)}) \\ \quad + \sum_{j=i+2}^{k-1} w_{\sigma(j)} \max(0, \sum_{q=1}^{j} p_{\sigma(q)} + p_{\sigma(k)} - p_{\sigma(i+1)} - d_{\sigma(j)}) \\ \quad + w_{\sigma(i+1)} \max(0, \sum_{q=1}^{k} p_{\sigma(q)} - d_{\sigma(i+1)}) \} \end{cases} \end{cases} \tag{3.1.1}$$

For job set N, it is easy to see that σ_n and $G(\sigma_n)$ can be obtained from the dynamic programming algorithm in $O(n^3)$ time.

3.1.3 The Modified Dynasearch for $1|r_i|\sum w_i T_i$

The idea of dynamic programming algorithm proposed by Congram et al. [12] can be used for problem $1|r_i|\sum w_i T_i$ too. The modified dynasearch takes the

same time complexity $O(n^3)$, while the size of the neighborhood is usually smaller than $2^{n-1}-1$. In this section, we give detailed introduction to show how we construct the modified dynasearch for problem $1|r_i|\sum w_iT_i$.

Suppose the initial given sequence is $\sigma = (\sigma(1), ..., \sigma(n))$. Similarly to the original dynamic programming algorithm, see Section 3.1.2, the new sequence σ' is built by the modified dynamic programming algorithm in a forward enumeration mode. In this modified algorithm, we build the new sequence from the 1 job partial sequence to the n-job sequence. There are $n+1$ states in the modified algorithm too.

Different from the original dynasearch algorithm, in state k, $k \in 0, 1, ..., n$, the modified dynasearch searches for the good *independent* swaps of the new partial sequence $(\sigma(1), ..., \sigma(k))$. The reason is that, for problem $1|r_i|\sum w_iT_i$, different permutations of σ_k generate different completion time $C(\sigma_k)$. To get a good upper bound, we also need to consider the right shift of jobs after σ_k. In state k, the order of the right shift sequence is fixed as $(\sigma(k+1), ..., \sigma(n))$. So the choice of sequence σ_k is possibly not the *best*. The search strategy of the modified dynamic programming algorithm is not the $best-improve$. In a word, the modified dynasearch searches for good but not the best *independent* swaps in the neighborhood. That is the reason why the neighborhood size of the modified dynasearch is usually smaller than $2^{n-1}-1$.

Suppose for $i = 0, ..., k-1$, where $k \in \{0, ..., n\}$, we have found and stored some information of state (i, σ), such as the sequence σ_i and the total weighted tardiness $TWT(\sigma_i)$. Here σ_i is a permutation of the partial sequence $(\sigma(1), ..., \sigma(i))$ and contains the good *independent* swaps for those jobs. Then in state (k, σ), finding σ_k relies not only on the information of previous states from $(0, \sigma)$ to $(k-1, \sigma)$, but also the right shift of jobs after job $\sigma(k)$. Following is the detailed steps of the modified dynasearch.

Procedure SMK. [12]

Step 1. Suppose we have found and stored σ_i and $TWT(\sigma_i)$ for $i = 0, ..., k-1$, for some $k \in \{0, ..., n\}$.

Step 2. Let $j = k-1$. Let $\sigma_k = (\sigma_j, \sigma(k)), S = (\sigma_k, \sigma(k+1), ..., \sigma(n))$ and compute the total weighted tardiness $TWT(S)$.

Step 3. Let $j = j-1$.

Step 4. If $j \geq 0$, construct the sequence $\sigma'_k = (\sigma_j, \sigma(j+1), \sigma(j+2), ..., \sigma(k-2), \sigma(k))$, then swap job $\sigma(k)$ and job $\sigma(j+1)$ in sequence σ'_k. Let $S' = (\sigma'_k, \sigma(k+1), ..., \sigma(n))$ and compute the total weighted tardiness $TWT(S')$; else goto Step 7.

Step 6. If $TWT(S') < TWT(S)$, set $\sigma_k = \sigma'_k$. Goto Step 3.

Step 7. Return σ_k and $TWT(\sigma_k)$. Stop.

Following is the implementation for the modified dynasearch in our program. A dummy job 0 is needed. For job 0, we have $r_0 = w_0 = p_0 = d_0 = 0$. In the implementation, for any given partial sequence σ of N, where $|\sigma| \leq n$, the total weighted tardiness of the partial sequence $TWT(\sigma)$ can be computed in $O(n)$ time.

Implementation of The Modified Dynasearch.

Step 1. Suppose the initial given sequence is $(\sigma(0), \sigma(1), ..., \sigma(n))$. Let $TWT_0 = 0, \sigma_0 = (\sigma(0)), \sigma_1 = (\sigma(0), \sigma(1))$ and $TWT_1 = TWT(\sigma_1)$.

Step 2. Let $TWT_2 =$

$$\min \begin{cases} TWT(\sigma_0, \sigma(2), \sigma(1), \sigma(3), ..., \sigma(n)), \\ TWT(\sigma_1, \sigma(2), \sigma(3)), ..., \sigma(n)) \end{cases}$$

and record the best partial sequence as σ_2.

Step 3. Let $TWT_3 =$

$$\min \begin{cases} TWT(\sigma_0, \sigma(3), \sigma(2), \sigma(1), \sigma(4), ..., \sigma(n)), \\ TWT(\sigma_1, \sigma(3), \sigma(2), \sigma(4), ..., \sigma(n)), \\ TWT(\sigma_2, \sigma(3), \sigma(4), ..., \sigma(n)) \end{cases}$$

and record the best partial sequence as σ_3.

...

Step n. Let $TWT_n =$

$$\min \begin{cases} TWT(\sigma_0, \sigma(n), \sigma(2), ..., \sigma(n-1), \sigma(1)), \\ TWT(\sigma_1, \sigma(n), \sigma(3), ..., \sigma(n-1), \sigma(2)), \\ ... \\ TWT(\sigma_{n-2}, \sigma(n), \sigma(n-1)), \\ TWT(\sigma_{n-1}, \sigma(n)) \end{cases}$$

and record the best sequence as σ_n. Return σ_n and TWT_n.

3.1.4 Implementation of Upper Bound I

In [12], their iterated dynasearch works as follows: the iterated dynasearch starts from a given initial sequence. After each iteration, they perform some prespecified type random swaps in the current new sequence and then start a new dynasearch with the new sequence. The iterated dynasearch stops after a total number of iterations. The number they choose is 100.

In our program, we don't perform any random swap in the new sequence after each iteration. Following is the implementation of the first upper bound.

Implementation of Upper Bound I.

Step 0. Suppose the given initial sequence is renumbered as $(1, ..., n)$. Let $S = (1, ..., n), S' = S, j = 0$ and sort S' in non-decreasing release date order. /*— S, S', S'' are arrays; S is used to contain the final sequence —*/

Step 1. Set $D = \{TWT(S')\}, K = 100, i = 0$ and $S'' = S'$. /*— D is a set; S' contains the temporary result —*/

Step 2. If $i < K$, set $i = i + 1$, apply the modified dynasearch to S'' and record the new solution sequence as S''; else goto Step 5. /*— apply dynasearch to S'' and get a new sequence —*/

Step 3. If $TWT(S'') \in D$, goto Step 5; else set $D = D \cup \{TWT(S'')\}$. /*— if D contains $TWT(S'')$, stop the iteration —*/

Step 4. If $TWT(S') > TWT(S'')$, let $S' = S''$. Goto Step 2. /*— if S'' is better than S', record it as S' —*/

Step 5. Let $j = j + 1$. If $TWT(S) > TWT(S')$, let $S = S'$. /*— update S —*/

Step 6. If $j = 2$, return S and $TWT(S)$. Stop. /*— if both orders are done, stop —*/

Step 7. Sort S' in earliest due date order and goto Step 1. /*— loop for the earliest due date order —*/

In the implementation, we apply the modified dynasearch to the sequence in non-decreasing release date order first and then to the sequence in earliest due date order separately. Using the two initial sequences makes the method generate better results in most cases.

3.2 Upper Bound II

3.2.1 Apparent Tardiness Cost Rule

There are many other heuristic algorithms for scheduling problems. Compared with the modified dynasearch, they need less time but the solution is weaker in most cases. In our program, we use the Apparent Tardiness Cost([34]) rule to give an upper bound for each node in the search tree. The modified dynasearch is only used for the root node in the search tree.

Under the Apparent Tardiness Cost rule, every time a job $i \in N$ is processed by the single machine, the new job j is selected to be processed next by an index function. Suppose the processed job set is N_1 and the completion time of job i is t. Then job j is selected by the following function [34]:

$$I_{i+1}(t) = \max_{j \in \{N-N_1\}} \frac{w_j}{p_j} exp\left(-\frac{\max\{d_j - p_j - t, 0\}}{K\bar{p}}\right).$$

Here K is a constant parameter and $\bar{p}$ is the average of the processing time for the job set $\{N - N_1\}$. Similarly to [34], we choose $K = 2$ in our program.

Chapter 4

Dominance Properties

In this chapter, we present the necessary background for the dominance properties used in the branch and bound method. We suggest a labeling scheme and the data structure. We then describe all the dominance properties and prove some of them.

4.1 Preliminaries

In this section, we introduce the background of the dominance properties and recall some definitions.

The framework of our depth first search is based on the branch and bound algorithm in [21]. Their branch and bound algorithm relies on ILOG CP optimizer, the ILOG constraint programming libraries. In the search tree of their branch and bound algorithm, the node is generated by functions of ILOG CP optimizer. At each node, there are two job sets of N: the scheduled job set N_1 and the unscheduled job set N_2. For each job $i \in N_1 \cup N_2$, we have a constraint variable $\bar{C}_i$, which is the completion time of job i. If the unscheduled job set N_2 of the node is empty, that means we obtain a job sequence of N and $\sum_{i=1}^{n} w_i \max(0, \bar{C}_i - d_i)$ is the total weighted tardiness of the sequence. The aim of the branch and bound algorithm is to find a job sequence of N and the value of $\sum_{i=1}^{n} w_i \max(0, \bar{C}_i - d_i)$ is minimum.

In [21], they use the $Edge - Finding$ [7] branching scheme. In this scheme, the algorithm sequences the jobs one by one on the single machine. At each node of the search tree, we choose a set of unscheduled jobs. Then the algorithm generates a new branch with a new node for every job in the selected job set. At each new node, one job from the selected job set is fixed in the first (or last) unscheduled position. With the depth first search strategy, their branch and bound algorithm sequences the jobs both from the left and from the right. There are several job sets at each node of the search tree. We recall them in the following definition.

Definition 4.1.1 [21]

At each node x in the search tree, let L_x be the scheduled partial sequence in the left. Let R_x be the scheduled partial sequence in the right. Let M_x be the unscheduled job set, i.e., $M_x = N - L_x - R_x$. Let $MF_x \subseteq M_x$ denote the job set such that every job $y \in MF_x$ can be fixed immediately after the last job of L_x. Let $ML_x \subseteq M_x$ denote the job set such that every job $y \in ML_x$ can be fixed immediately before the first job of R_x.

In the algorithm from [21], every time a new node x is added into the search tree, we filter some jobs out from MF_x by some dominance properties. Then the algorithm selects a job i from MF_x and branches to a new node y, in which job i is sequenced after L_x. We remove job i from MF_x when we backtrack from node y to node x. At node y, after filtering, if set $MF_y = \emptyset$ or $ML_y = \emptyset$, we backtrack from y to x. If there is only one job left in ML_y after filtering, then this job is immediately sequenced before the first job of R_y.

In this chapter, we have an assumption ([21]): at each node x of the search three, the release dates of jobs in M_x are adjusted according to the completion time $C(L_x)$. Suppose the completion time of L_x is $C(L_x)$, then for each job $i \in M_x$, we have $r_i = \max(r_i, C(L_x))$.

Before we introduce all the dominance properties in our branch and

bound algorithm, we recall some definitions.

Definition 4.1.2 [21]

For any two partial sequences σ_1 and σ_2 of N, if $\{i|i \in \sigma_1\} = \{j|j \in \sigma_2\}$ and one of the following conditions is satisfied, then the partial sequence σ_1 is said to be as good as σ_2.

1. *If $C(\sigma_1) \leq C(\sigma_2)$ and $TWT(\sigma_1) \leq TWT(\sigma_2)$.*
2. *If $C(\sigma_1) > C(\sigma_2)$ and $TWT(\sigma_1) + (\max(C(\sigma_1), \hat{r}) - \max(C(\sigma_2), \hat{r}))$ $* \sum_{l \in N - \{i|i \in \sigma_1\}} w_l \leq TWT(\sigma_2)$, where $\hat{r} = \min_{\{l \in N - \{i|i \in \sigma_1\}\}} r_l$.*

Based on Definition 4.1.2, we have the following definition of *better* sequence.

Definition 4.1.3 [21]

For any two partial sequences σ_1 and σ_2, if $\{i|i \in \sigma_1\} = \{j|j \in \sigma_2\}$ and one of the following conditions is satisfied, then the partial sequence σ_1 is said to be better than σ_2.

1. *If σ_1 is as good as σ_2 and σ_2 is not as good as σ_1.*
2. *If σ_1 is as good as σ_2, σ_2 is as good as σ_1 and σ_1 is smaller than σ_2 by the lexicographic order.*

Most of our dominance properties are based on Definition 4.1.3.

4.2 Dominance Property From Visited Nodes

In this section, we first introduce a data structure. Then we describe how we use it within a dominance property.

4.2.1 The Red-Black Tree

The *red-black tree* is a type of *height-balanced binary search tree*, a data structure proposed by Guibas and Sedgewick [18].

The *binary tree* is an acyclic connected graph such that each node has zero or at most two children nodes. The node without a parent node is the root node. Except the root node, each node has a parent node. Each node contains a unique key, for example, an integer. Besides the key, each node contains pointers pointing to its children and parent. If a node does not have a child or the parent, the value of the corresponding pointer is NIL. A *leaf* node has no children. The *height* of a *binary tree* refers to the size of the longest path over all possible paths from the root node to a *leaf* node.

A *binary search tree* is a *binary tree* and every node in the tree is subject to the following constraints:

- *1. All the keys of the left subtree for the node are less than the key of the node.*
- *2. All the keys of the right subtree for the node are greater than the key of the node.*

For a given set of integers H, it is clear that we can construct lots of corresponding *binary search trees.* Among all those *binary search trees*, a *height-balanced binary search tree* refers to the *binary search tree* with the minimum *height.* It is easy to see that the *height* of a *height-balanced binary search tree* for set H is bounded by $\log_2(|H|)$.

A *red-black tree* is a *binary search tree* with the extra color property per node and the tree is subject to the following constraints([14]):

- *1. Each node is either black or red.*
- *2. The root node is black.*
- *3. Each leaf node is black and doesn't contain a key.*
- *4. For each red node, both children are black.*
- *5. For each node, every path from the node to any of its descendant leaves contains the same number of black nodes.*

If a node is black, there is no restriction on the colors of its children.

There is no binding relation between the key and the color of any node. Because of the five constraints [14], a *red-black tree* is approximately *height-balanced.* We have the following result about the height of a red-black tree.

Lemma 4.2.1 [14]

A red-black tree with n non-leaf nodes has height at most $2\log_2(n+1)$.

Suppose a *red-black tree* contains n keys. For a given key, we can search in the *red-black tree* to see whether it exists or not. The search begins at the root node, pointed by a pointer variable p. We compare the given key with the key of the node pointed by p. If they are equal, we stop the search. If the given key is smaller (or greater), we set p to its left (or right) child and continue the search recursively until either a leaf node is met or the given key is found. The read-only operation doesn't modify the red-black tree.

If the red-black tree doesn't contain the key, we can create a new red node with the given key and add it into the tree. The insert operation modifies the tree and possibly makes it violate the five constraints. Extra work is needed to restore the five constraints. First, we search in the tree to find the parent node for the new red node by key comparison. Second, we repaint the colors and change the pointers of the new node and some other nodes according to their colors. In [14], they show that read-only and insert operations for a given key in the *red-black tree* both take $O(\log_2(n))$ time. For the detailed procedures of the operations, see [14].

4.2.2 The Red-Black Tree and The Dominance Property

Problem $1||\sum w_jT_j$ is a special case of problem $1||\sum h_j(C_j)$, where $h_j(C_j)$ is a function of C_j for $j = 1, ..., n$. In [26], there is a dynamic programming algorithm for problem $1||\sum h_j(C_j)$. The dynamic programming algorithm can

be described by a recursive function([26]):

$$V(\{1,...,n\}) = \begin{cases} V(\{j\}) = h_j(p_j), j = 1,...,n \\ V(J) = \min_{j \in J}(V(J - \{j\}) + h_j(\sum_{k \in J} p_k)) \end{cases}$$

In this recursive function, J is a subset of N. To find $V(N)$, we need to compute all $V(J)$. If $|J| = k \leq n$, then there are $n!/(k!(n-k)!)$ subsets of cardinality k. The number of all the subsets is 2^n.

In [21], they use the no-good recording technique [35] as a dominance property. The idea of this technique is similar to the above dynamic programming algorithm. During the searching process of the branch and bound method, every time a new node x is added into the search tree, for every $i \in MF_x$, we use the bit set of the scheduled partial sequence (L_x, i) to query in the no-good list. Suppose there exists a visited node y in the no-good list such that $\{k|k \in (L_x, i)\} = \{j|j \in L_y\}$. Node y contains data of the completion time $C(L_y)$ and the weighted tardiness $TWT(L_y)$. Then we know which partial sequence is better according to Definition 4.1.3. If L_y is better, we remove i from MF_x; otherwise, we add some information of the partial sequence (L_x, i) into the no-good list.

In our branch and bound algorithm, the no-good list is a red-black tree, see Section 4.2.1. We use the standard binary labeling scheme [30] for all the feasible subsets. In the standard binary labeling scheme, the information of a job set is stored in a bit array. In the bit array for a job set, if the value of bit $i, i \in N$ is 1, then this means job i is in the job set, while 0 means not. For a given feasible subset, we use its bit array as the key to perform read-only and insert operations in the red-black tree.

Besides the unique bit array of a visited partial sequence σ, each node of the red-black tree contains data of the completion time $C(\sigma)$ and the weighted tardiness $TWT(\sigma)$. Because the maximum number of the bit arrays of feasible subsets for the problem is 2^n, the height of the red-black tree is bounded by $2\log_2(2^n + 1)$. Considering the second condition of Definition 4.1.2, we

maintain a list of pairs of $(C(\sigma), TWT(\sigma))$ at each red-black tree node. In most cases, it takes $O(\log_2(2^n)) = O(n)$ time to perform the read-only and insert operations in the red-black tree.

4.3 Dominance Property By Release Date and Processing time

A sequence is *active* if it is impossible for us to find a new sequence by changing the job order in the given sequence such that there is at least one job that is finished earlier and no job is delayed. In [4, 13], they show that the set of *active* sequences is dominant. We have the following result for the unscheduled job set.

Theorem 4.3.1 [4, 13]

Given a new node x in the search tree, for any job $k, k \in M_x$, if $r_k \geq \min_{\{i \in M_x\}}\{r_i + p_i\}$, then $MF_x = MF_x - \{k\}$.

This dominance property filters some jobs out from the unscheduled job set based only on the release date and the processing time.

4.4 Dominance Property From Local Optimality

In [11, 9, 8, 10], Chu proposes a local optimality condition for a single machine problem with only two adjacent jobs. The local optimality condition can be applied to the total weighted tardiness problem as follows.

Theorem 4.4.1 [11, 21]

Suppose that in a single machine scheduling problem, there are two adjacent jobs j and k to be processed by the machine at time t. It is optimal to sequence job k after job j if and only if $TWT_{jk}(t) \leq TWT_{kj}(t)$, where $TWT_{jk}(t) = w_j \max(0, \max(r_j, t) + p_j - d_j) + w_k \max(0, \max(\max(r_j, t) +$

$p_j, \max(r_k, t)) + p_k - d_k)$ *and* $TWT_{kj}(t) = w_k \max(0, \max(r_k, t) + p_k - d_k) + w_j \max(0, \max(\max(r_k, t) + p_k, \max(r_j, t)) + p_j - d_j)$.

Then in [21], the authors modify this result to find a dominant subset for problem $1|r_i| \sum w_i T_i$.

Definition 4.4.2 [21]

A job sequence σ is said to be LO-Active (Locally Optimal Active), if every adjacent pair of jobs j and k (in which job j precedes k) satisfies at least one of the following conditions:

1. $\max(r_j, C_{j-1}(\sigma)) < \max(r_k, C_{j-1}(\sigma))$

2. $TWT_{jk}(C_{j-1}(\sigma)) \leq TWT_{kj}(C_{j-1}(\sigma))$.

Theorem 4.4.3 [21]

If a sequence is optimal for problem $1|r_i| \sum w_i T_i$, it is LO-Active.

Based on Definition 4.4.2 and Theorem 4.4.3, we have the following results.

Definition 4.4.4 [21]

A job sequence σ is said to be LOWS-Active (Locally Optimal Well Sorted Active), if every adjacent pair of jobs j and k (in which job j precedes k) satisfies at least one of the following conditions:

1. $TWT_{jk}(\delta) < TWT_{kj}(\delta)$

2. $TWT_{jk}(\delta) = TWT_{kj}(\delta)$ *and* $\max(r_j, \delta) \leq \max(r_k, \delta)$

3. $TWT_{jk}(\delta) > TWT_{kj}(\delta)$ *and* $\max(r_j, \delta) < \max(r_k, \delta)$

where $\delta = C_{j-1}(\sigma)$.

Not all LOWS-Active sequences are optimal. In [21], the authors use the following two results as local dominance properties in the branch and bound algorithm.

Theorem 4.4.5 [21]

The subset of LOWS-Active sequences is dominant for problem $1|r_i| \sum w_i T_i$.

Theorem 4.4.6 [21]

Given a new node x in the search tree, suppose job j is the last job of the partial sequence L_x. For every job $k \in MF_x$, k can be removed from MF_x if at least one of the following conditions is satisfied:

1. $\max(r_k, \delta) < \max(r_j, \delta)$ *and* $TWT_{kj}(\delta) \leq TWT_{jk}(\delta)$

2. $\max(r_k, \delta) \leq \max(r_j, \delta)$ *and* $TWT_{kj}(\delta) < TWT_{jk}(\delta)$

3. $\max(r_k, \delta) = \max(r_j, \delta)$, $TWT_{kj}(\delta) = TWT_{jk}(\delta)$ *and* $j < k$

where $\delta = C_{j-1}(L_x)$.

4.5 Dominance Properties From the Scheduled Partial Sequence

Compared with the dominance rules introduced in the above sections, we have a more general result.

Theorem 4.5.1 [21]

Given a new node x in the search tree with the scheduled partial sequence L_x, for every job $i \in MF_x$, i can be removed from MF_x if there exists another partial sequence L_y such that $\{k|k \in (L_x, i)\} = \{j|j \in L_y\}$ and L_y is better than (L_x, i).

4.6 Dominance Properties Based on Unscheduled Jobs

In this section, we recall two important dominance properties based on job interchange and insertion. The dominance properties introduced in this section only take into account information for unscheduled jobs.

4.6.1 Dominance Properties By Interchange and Insertion

Given any two jobs j and k in the unscheduled job set M_x, we recall under which condition [21] job j dominates job k in the first unscheduled position. For the situation of job interchange between j and k, we have the assumption: $w_j \geq w_k$.

At a new node x, L_x is the partial sequence fixed in the left and R_x is the partial sequence scheduled in the right. Suppose a feasible partial sequence of unscheduled job set M_x is given by (k, A_x, j, B_x). Here A_x is the partial sequence between job k and j and B_x is the partial sequence between job j and sequence R_x. We get the full sequence $\sigma = (L_x, k, A_x, j, B_x, R_x)$. After exchanging job k and job j in σ, we get the new sequence $\sigma' = (L_x, j, A_x, k, B_x, R_x)$. Let $TWT(\sigma)$ denote the total weighted tardiness of σ and $WT(k)$ denote the weighted tardiness of job k. If $TWT(\sigma') \leq TWT(\sigma)$, then job j dominates job k in the first unfixed position. Thus, job k can be removed from MF_x [21].

We can expand the total weighted tardiness of the two sequences σ and σ' to:

$$\begin{cases} TWT(\sigma) = TWT(L_x) + WT(k) + TWT(A_x) \\ \qquad\qquad +WT(j) + TWT(B_x) + TWT(R_x) \\ \\ TWT(\sigma') = TWT(L_x) + WT(j') + TWT(A'_x) \\ \qquad\qquad +WT(k') + TWT(B'_x) + TWT(R'_x) \end{cases}$$

Then $TWT(\sigma) - TWT(\sigma')$ can be transformed to:

$$\begin{aligned} &TWT(\sigma) - TWT(\sigma') = \\ &\quad WT(k) + WT(j) - (WT(j') + WT(k')) + && (4.6.1) \\ &\quad TWT(A_x) + TWT(B_x) - (TWT(A'_x) + TWT(B'_x)) + && (4.6.2) \\ &\quad TWT(R_x) - TWT(R'_x) && (4.6.3) \end{aligned}$$

To get an estimation for $TWT(\sigma) - TWT(\sigma')$, we first recall the following definitions.

Definition 4.6.1 [21]

Let $s_j(\sigma)$ denote the start time of job j in sequence σ and $s_k(\sigma')$ be the start time of job k in sequence σ'. Let $C_k(\sigma) = r_k + p_k$ denote the completion time of job k in sequence σ and $C_j(\sigma') = r_j + p_j$ be the completion time of job j in sequence σ'. Let $\phi_1 = s_k(\sigma') - s_j(\sigma)$ and ϕ_2 be the maximum shift of job set $A_x \cup B_x$ in σ', i.e., $\phi_2 = \max(0, C_j(\sigma') - C_k(\sigma), C_k(\sigma') - C_j(\sigma))$. Let ϕ_3 be the maximum shift of job set R_x in σ', i.e., $\phi_3 = \max(0, C_k(\sigma') - C_j(\sigma))$.

By the above definitions of ϕ_1,ϕ_2 and ϕ_3 from [21], we get an estimation for (4.6.1),(4.6.2) and (4.6.3) [21]:

$$TWT(\sigma) - TWT(\sigma') \geq \Phi_{jk}(s_j(\sigma), \phi_1, \phi_2, \phi_3), \tag{4.6.4}$$

where function $\Phi_{jk}(s, \phi_1, \phi_2, \phi_3)$ is defined as [21]:

$$\begin{aligned}\Phi_{jk}(s, \phi_1, \phi_2, \phi_3) = {} & w_j \max(0, s + p_j - d_j) - w_k \max(0, s + \phi_1 + p_k - d_k) \\ & + w_k \max(0, r_k + p_k - d_k) - w_j \max(0, r_j + p_j - d_j) \\ & - \phi_2 \textstyle\sum_{\{l \in A_x \cup B_x\}} w_l - \phi_3 \sum_{\{l \in R_x\}} w_l\end{aligned} \tag{4.6.5}$$

It is easy to see that $\Phi_{jk}(s_j(\sigma), \phi_1, \phi_2, \phi_3)$ gives a lower bound estimation for $TWT(\sigma) - TWT(\sigma')$ by (4.6.4).

The following results show under what conditions, we can remove job k from MF_x.

Lemma 4.6.2 [21]

For given ϕ_1, ϕ_2 and ϕ_3, $\Phi_{jk}(s, \phi_1, \phi_2, \phi_3)$ is a function of s. If $w_j \geq w_k$, Φ_{jk} is nondecreasing on $[d_j - p_j, +\infty]$ and obtains its global minimum at $s = d_j - p_j$.

Theorem 4.6.3 [21]

Given a new node x in the search tree, for any two jobs $j, j \in M_x$ and $k, k \in MF_x$, if $w_j \geq w_k$ and $\hat{\Phi}_{jk}(\phi_1, \phi_2, \phi_3) \geq 0$, then job j domi-

nates job k in the first position after the scheduled partial sequence L_x, where $\hat{\Phi}_{jk}(\phi_1, \phi_2, \phi_3) = \Phi_{jk}(\max(d_j - p_j, r_k + p_k), \phi_1, \phi_2, \phi_3)$.

For the case of job insertion, we construct σ and σ' as follows

$$\begin{cases} \sigma = (L_x, k, A_x, j, B_x, R_x) \\ \sigma' = (L_x, j, k, A_x, B_x, R_x) \end{cases}$$

With a same analysis as for interchange, we have the following result for insertion.

Theorem 4.6.4 [21]

Given a new node x in the search tree, for any two jobs $j, j \in M_x$ and $k, k \in MF_x$, if $\Phi_{jk}(r_k+p_k, r_j+p_j-r_k-p_k, r_j+p_j-r_k, \max(0, r_j-r_k)) \geq 0$, then job j dominates job k in the first position after the scheduled partial sequence L_x.

4.6.2 Eight Cases of Interchange

In this section, we give all possible eight cases [21] to show how to compute the values of ϕ_1, ϕ_2 and ϕ_3 introduced in the last section. For the detailed discussion, see [21]. Figure 4.1 gives a tree of different conditions of the eight cases. Table 4.1 shows values of ϕ_1, ϕ_2 and ϕ_3 for the eight cases.

Eight Cases.[21]

Case 1. If $r_j + p_j \leq r_k + p_k$ and $p_j < p_k$, then $\phi_1 = 0, \phi_2 = \phi_3 = p_k - p_j$.

Case 2. If $r_j+p_j \leq r_k+p_k, p_j < p_k, r_k+p_j < \max_{\{i \in M_x\}}\{r_i\} < r_k+p_k$, and $r_j \leq r_k$, then $\phi_1 = \max_{\{i \in M_x\}}\{r_i\} - r_k - p_k, \phi_2 = \phi_3 = \max_{\{i \in M_x\}}\{r_i\} - r_k - p_j$.

Case 3. If $r_j+p_j \leq r_k+p_k, p_j < p_k, r_j \leq r_k$, and $\max_{\{i \in M_x\}}\{r_i\} < r_k+p_j$, then $\phi_1 = \max(r_j + p_j, \max_{\{i \in M_x\}}\{r_i\}) - r_k - p_k, \phi_2 = \phi_3 = 0$.

Table 4.1: Eight cases of interchange [21]

Case	Values of ϕ_1, ϕ_2, ϕ_3
1	$\phi_1 = 0, \phi_2 = \phi_3 = p_k - p_j$
2	$\phi_1 = \max_{\{i \in M_x\}}\{r_i\} - r_k - p_k, \phi_2 = \phi_3 = \max_{\{i \in M_x\}}\{r_i\} - r_k - p_j$
3	$\phi_1 = \max(r_j + p_j, \max_{\{i \in M_x\}}\{r_i\}) - r_k - p_k, \phi_2 = \phi_3 = 0$
4	$\phi_1 = \max(r_j + p_j, \max_{\{i \in M_x\}}\{r_i\}) - r_k - p_k,$ $\phi_2 = \phi_3 = \max(r_j + p_j, \max_{\{i \in M_x\}}\{r_i\}) - r_k - p_j$
5	$\phi_1 = \phi_2 = \phi_3 = 0$
6	$\phi_1 = \max(r_j + p_j, \max_{\{i \in M_x\}}\{r_i\}) - r_k - p_k, \phi_2 = \phi_3 = 0$
7	$\phi_1 = r_j + p_j - r_k - p_k, \phi_2 = \phi_3 = r_j - r_k$
8	$\phi_1 = r_j + p_j - r_k - p_k, \phi_2 = r_j + p_j - r_k - p_k, \phi_3 = \max(0, r_j - r_k)$

Case 4. If $r_j + p_j \leq r_k + p_k, p_j < p_k, \max_{\{i \in M_x\}}\{r_i\} < r_k + p_k$, *and* $r_j > r_k$, *then* $\phi_1 = \max(r_j + p_j, \max_{\{i \in M_x\}}\{r_i\}) - r_k - p_k, \phi_2 = \phi_3 = \max(r_j + p_j, \max_{\{i \in M_x\}}\{r_i\}) - r_k - p_j$.

Case 5. If $r_j + p_j \leq r_k + p_k, p_j \geq p_k$, *then* $\phi_1 = \phi_2 = \phi_3 = 0$.

Case 6. If $r_j + p_j \leq r_k + p_k, p_j \geq p_k$ *and* $\max_{\{i \in M_x\}}\{r_i\} < r_k + p_k$, *then* $\phi_1 = \max(r_j + p_j, \max_{\{i \in M_x\}}\{r_i\}) - r_k - p_k, \phi_2 = \phi_3 = 0$.

Case 7. If $r_j + p_j > r_k + p_k, p_j < p_k$, *then* $\phi_1 = r_j + p_j - r_k - p_k, \phi_2 = \phi_3 = r_j - r_k$.

Case 8. If $r_j + p_j > r_k + p_k, p_j \geq p_k$, *then* $\phi_1 = r_j + p_j - r_k - p_k, \phi_2 = r_j + p_j - r_k - p_k$ *and* $\phi_3 = \max(0, r_j - r_k)$.

4.7 Some Other Dominance Rules

In this section, we give some other dominance rules for problem $1|r_i|\sum w_i T_i$.

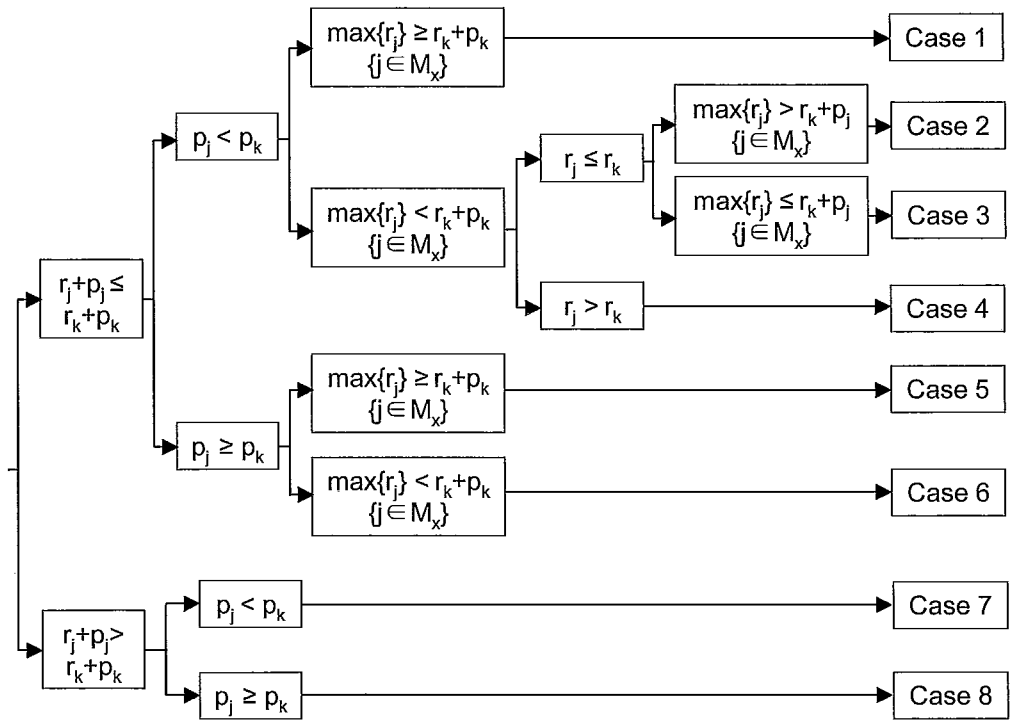

Figure 4.1: Tree of the eight cases [21]

Theorem 4.7.1 [9, 2]

Given a new node x in the search tree, for any two jobs j and k in M_x, if $w_j \geq w_k, p_j \geq p_k, r_j + p_j \leq r_k + p_k$, and $d_j \leq d_k$, then job j dominates job k in the first position after the scheduled partial sequence L_x.

Theorem 4.7.2 [9, 2, 6]

Given a new node x in the search tree, for any two jobs j and k in M_x, job j dominates job k in the first position after the scheduled partial sequence L_x if all the following conditions are satisfied: 1. $r_j \leq r_k, d_j \leq d_k, w_j \geq w_k$, and $r_j + p_j \leq r_k + p_k$, 2. $\max(r_j, r_k) \leq \min_{i \in M_x}\{r_i + p_i\}$.

Theorem 4.7.3 [6]

Given a new node x in the search tree, for any jobs $j \in M_x$ and $k \in MF_x$, if $w_j \geq w_k, r_j + p_j \leq r_k + p_k$ and $w_j(r_k + p_k - r_j - p_j) + w_j p_j - w_k p_k \geq (p_k - p_j) \sum_{\{l \in \{M_x - \{j,k\} \cup R_x\}\}} w_l$, then job j dominates job k in the first position after the scheduled partial sequence L_x.

Theorem 4.7.4 [6]

Given a new node x in the search tree, for any jobs $j \in M_x$ and $k \in MF_x$, if $w_j \geq w_k, r_j + p_j \geq r_k + p_k$, and $w_j p_j - w_k p_k \geq (r_j + p_j - r_k - p_k) \sum_{\{l \in M_x \cup R_x\}} w_l + \max(0, p_k - p_j) \sum_{\{l \in M_x - \{j,k\} \cup R_x\}} w_l$, then job j dominates job k in the first position after the scheduled partial sequence L_x.

Theorem 4.7.5 [6]

Given a new node x in the search tree, suppose the scheduled partial sequence is $L_x = (L'_x, k, y)$. For any job j in MF_x, if there are no inserted idle times in sequences (j, y, k) and (k, y, j), then sequence (L'_x, j, y, k) is better than sequence (L'_x, k, y, j) such that $(r_k + p_k - r_j - p_j)(w_j + w_k + w_y) + (w_j - w_k)(p_j + p_k + p_y) + w_k p_j - w_j p_k \geq \max(0, r_j - r_k) \sum_{\{l \in M_x - \{j\}\}} w_l$.

Theorem 4.7.6 [10]

Given a new node x in the search tree, for any job j in MF_x, j can be removed from MF_x if there exists a job $k \in L_x$ such that $\max(r_j, C_{k-1}(L_x)) + p_j \leq \max(r_k, C_{k-1}(L_x)) + p_k$ *and* $\max(r_j, C_{k-1}(L_x)) - \max(r_k, C_{k-1}(L_x)) \leq (p_j - p_k)(|M_x| - 1)$.

Theorem 4.7.7 [10]

Given a new node x in the search tree, for any job j in MF_x, job j can be removed from MF_x if there exists a job $k \in L_x$ such that $p_j \geq p_k$ and $p_j - p_k \geq (\max(r_j, C_{k-1}(L_x))) + p_j - \max(r_k, C_{k-1}(L_x))) - p_k)(|L_x| - i + 2)$, where $k = L_x(i), 1 \leq i \leq |L_x|$.

Actually, the above dominance rules are included in Theorem 4.5.1 and Theorem 4.6.3. For the detailed proof of following results, see [20].

Proposition 4.7.8 [21]

Theorems from 4.7.1 to 4.7.4 are included in Theorem 4.6.3.

Proposition 4.7.9 [21]

Theorems from 4.7.5 to 4.7.7 are included in Theorem 4.5.1.

4.8 Applying Dominance Properties

Every time a new node x is added into the search tree, the possible first job set MF_x is equal to the unscheduled job set M_x. Then we apply dominance properties one by one to filter out as many jobs in MF_x as possible.

Theorem 4.3.1 is the first dominance rule [21] applied to MF_x. If the distribution of unscheduled jobs are not dense in release date, Theorem 4.3.1 can filter out most of them.

For every pair of jobs in the possible first job set MF_x, we apply Theorem 4.6.3 and 4.6.4. By inserting and interchanging [21], we also generate the

permutations of i and jobs in L_x. If there exists a better permutation of (L_x, i), then i can be removed from MF_x.

In [21], for every job $i \in MF_x$, they generate all permutations of last 5 jobs in L_x and i. If there exists a better permutation of the 6 jobs, then i can be removed from MF_x.

Finally, for each job i in MF_x, we use the bit set of (L_x, i) to query in the visited no-good list [21]. According to Theorem 4.5.1, if there exists a visited partial sequence which is better than (L_x, i), we can remove i from MF_x; otherwise, we need to add some information into the no-good list for future queries.

Chapter 5

Computational Results

In this chapter we provide the computational results for the branch and bound algorithm. Our program is based on the solver from Jouglet et al. [21]. We add the new lower bound and the new upper bound modules. We rewrite the old lower bound and modify the dominance rules modules. Codes are written in C++ and all computations are performed on a Windows PC with Intel Core2Duo 2.40GHz processor and 1Gb RAM. All running times are reported in seconds.

The generating scheme of the test instances is from [9]. Suppose the input size of the problem is $|N| = n$. For each job $i \in N$, the processing time p_i and the weight w_i are positive integers randomly chosen from $[1, 10]$. The release date r_i is an integer randomly chosen from $[0, \alpha \sum p_i]$, where α is a float parameter from $\{0.0, 0.5, 1.0, 1.5\}$. The due date d_i is an integer randomly chosen from $[(r_i + p_i), (r_i + p_i) + \beta \sum p_i]$, where β is a float parameter from $\{0.05, 0.25, 0.5\}$. There are 12 pairs of (α, β). For each pair of (α, β), ten instances are generated randomly. For a $|N| = n$ jobs problem, we generate 120 test instances. Test sets are generated for $n = 30, 35, 40, 45, 50, 60$. In total, there are 720 test instances. For every test instance, we set up one hour time limit for the program.

First, we report the computational results of the new solver and the old

solver in one table for instances with 30 jobs to 60 jobs. Table 5.1 reports the total number and the percent of instances solved by the two solvers. For the hardest instances, the new solver can handle up to 40 jobs in one hour time limit, while this number for the old solver is 30. For the 60-job problem, the old solver is able to solve 74.2% of all the test instances, and the percentage of instances solved by the new solver is 92.5%.

Second, considering the maximum problem size of all instances solved by the new solver is $n = 40$, we give the specialized computational results for 40-job problems. Figure 5.1 shows the distribution of solution times for the instances of the 40-job problems. Source data are from Table 5.7 with the new solver. By Figure 5.1, we know that instances from $(0, 0.5), (0.5, 0.05), (0.5, 0.25)$ and $(0.5, 0.5)$ are hard to solve. And $(\alpha = 0.5, \beta = 0.5)$ is the hardest parameter setting. Table 5.2 reports the percent of instances solved by the two solvers for hard instances with the parameter sets of $(0, 0.5), (0.5, 0.05), (0.5, 0.25)$ and $(0.5, 0.5)$.

Table 5.3 gives the comparison of the two lower bound schemes for 40-job instances. For each pair of (α, β), we report the number of times LB_1 is greater than LB_2, the number of times LB_2 is greater than LB_1 and the average difference. In most cases, the new lower bound scheme (LB_1) brings us a better lower bound. In Table 5.4, we report the number of times UB_1 is smaller than UB_2, the number of times UB_2 is smaller than UB_1, the average relative deviation (ARD) and the maximum relative deviation (MRD) of the two upper bound schemes. If UB is the value given by an upper bound scheme and OPT is the optimal value for a test instance, then the value of relative deviation is $(UB - OPT)/OPT$.

Finally, for each set of the 120 test instances from 30-job problems to 60-job problems, we give the computational results of the two solvers in a table. From Table 5.5 to 5.10, we report the number of unsolved instances,

Table 5.1: Comparison of the new and old solver and percentage solved.

	New		*Old*	
n	*Solved*	*Perc.*	*Solved*	*Perc.*
30	120	100%	120	100%
35	120	100%	119	99.2%
40	120	100%	115	95.8%
45	118	98.3%	106	88.3%
50	115	95.8%	104	86.7%
60	111	92.5%	89	74.2%

Table 5.2: Percentage solved for hard pairs of (α, β).

(α, β)	$(0, 0.5)$		$(0.5, 0.05)$		$(0.5, 0.25)$		$(0.5, 0.5)$	
n	*New*	*Old*	*New*	*Old*	*New*	*Old*	*New*	*Old*
30	100%	100%	100%	100%	100%	100%	100%	100%
35	100%	100%	100%	100%	100%	100%	100%	90%
40	100%	100%	100%	100%	100%	100%	100%	50%
45	100%	90%	100%	100%	100%	70%	80%	0%
50	100%	100%	100%	100%	100%	40%	50%	0%
60	100%	0%	100%	90%	100%	0%	10%	0%

the average and the maximum running time for each pair of (α, β). It can be seen that the new solver is able to find the optimal schedule for more instances. Furthermore, it requires substantially less time to solve most instances. It is easy to see that the relation between the instance hardness and the pair of (α, β) is not linear. If $\alpha = 0$ or $\alpha = 0.5$, the hardness of instance increases with β. For large values of α and β, because the distribution of the jobs is not dense, test instances are easier to solve.

Table 5.3: Comparison of the two lower bound schemes for 40-job instances.

α	β	$LB_1 > LB_2$	$Avg.Diff.$	$LB_2 > LB_1$	$Avg.Diff.$
0	0.05	486	33.7	7	11
0	0.25	14906	289.6	42	16.5
0	0.5	187051	977.3	720	42.3
0.5	0.05	32954	86.4	17410	90.7
0.5	0.25	363325	619.2	1013	54.3
0.5	0.5	12617816	252.2	304	37.9
1	0.05	29383	103.8	33299	98.0
1	0.25	42251	360.6	0	0
1	0.5	39	9.3	0	0
1.5	0.05	954	91.5	35	37.5
1.5	0.25	428	39.4	0	0
1.5	0.5	8	9	0	0

Table 5.4: Comparison of the two upper bound schemes for 40-job instances.

α	β	$UB_1 < UB_2$	ARD	MRD	$UB_2 < UB_1$	ARD	MRD
0	0.05	9	0.00	0.00	0	0.00	0.00
0	0.25	10	0.00	0.00	0	0.01	0.02
0	0.5	9	0.01	0.02	1	0.02	0.04
0.5	0.05	8	0.11	0.31	2	0.13	0.29
0.5	0.25	9	0.06	0.18	1	0.20	0.49
0.5	0.5	6	0.55	3.62	4	0.57	1.49
1	0.05	3	0.67	1.33	7	0.36	0.94
1	0.25	7	13.06	125	3	9.87	53
1	0.5	1	9.92	39	5	6.09	27
1.5	0.05	0	6.57	16.10	10	0.96	2.49
1.5	0.25	3	17.98	147	2	1.17	10
1.5	0.5	0	1.73	17.3	1	0.03	0.3

ARD: the average relative deviation
MRD: the maximum relative deviation

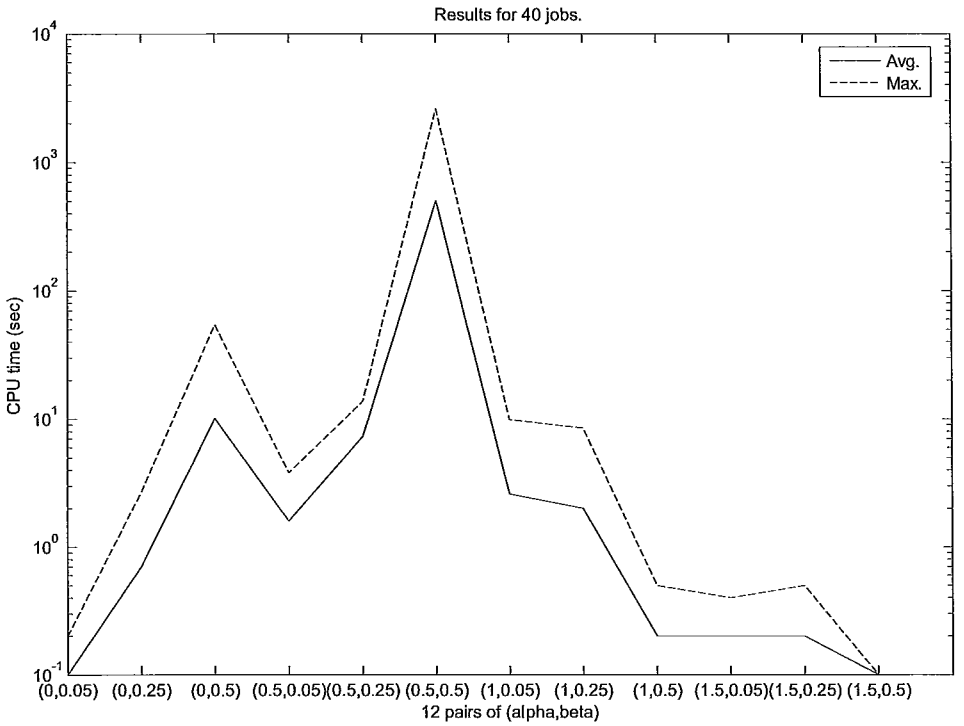

Figure 5.1: Hardness of 40-job instances for 12 pairs of (α, β).

Avg. : the average cpu time of the 10 instances for a pair of (α, β)
Max. : the maximum cpu time of the 10 instances for a pair of (α, β)

Table 5.5: Results for 30-job instances.

		New			*Old*		
α	β	*Avg.*	*Max.*	*U*	*Avg.*	*Max.*	*U*
0	0.05	0.1	0.1	-	0.1	0.1	-
0	0.25	0.2	0.6	-	1.2	3.0	-
0	0.5	1.0	4.4	-	10.6	31.8	-
0.5	0.05	0.2	0.3	-	0.5	1.0	-
0.5	0.25	1.1	5.8	-	24.0	108.2	-
0.5	0.5	14.0	40.7	-	65.9	201.4	-
1	0.05	0.1	0.4	-	0.2	0.5	-
1	0.25	0.1	0.2	-	0.1	0.2	-
1	0.5	0.1	0.1	-	0.1	0.1	-
1.5	0.05	0.1	0.2	-	0.1	0.1	-
1.5	0.25	0.1	0.1	-	0.1	0.1	-
1.5	0.5	0.1	0.1	-	0.1	0.1	-

U: the number of unsolved instances

Table 5.6: Results for 35-job instances.

		New			*Old*		
α	β	*Avg.*	*Max.*	*U*	*Avg.*	*Max.*	*U*
0	0.05	0.1	0.1	-	0.1	0.2	-
0	0.25	0.4	1.1	-	5.6	21.0	-
0	0.5	2.2	4.7	-	151.3	869.1	-
0.5	0.05	0.4	0.9	-	2.4	8.6	-
0.5	0.25	6.6	21.6	-	155.4	395.2	-
0.5	0.5	53.7	208.8	-	774.4	3600	1
1	0.05	0.28	0.9	-	0.2	0.6	-
1	0.25	0.3	2.0	-	0.3	2.4	-
1	0.5	0.1	0.3	-	0.1	0.5	-
1.5	0.05	0.2	0.3	-	0.1	0.2	-
1.5	0.25	0.1	0.1	-	0.1	0.1	-
1.5	0.5	0.1	0.1	-	0.1	0.1	-

U: the number of unsolved instances

Table 5.7: Results for 40-job instances.

		New			*Old*		
α	β	*Avg.*	*Max.*	*U*	*Avg.*	*Max.*	*U*
0	0.05	0.1	0.2	-	0.2	0.4	-
0	0.25	0.7	2.7	-	11.1	74.0	-
0	0.5	10.1	54.3	-	263.1	903.6	-
0.5	0.05	1.6	3.8	-	7.3	17.5	-
0.5	0.25	7.3	13.7	-	565.6	3284.5	-
0.5	0.5	502.7	2604.8	-	2301.8	3600	5
1	0.05	2.6	9.9	-	3.7	14.4	-
1	0.25	2.0	8.5	-	2.7	12.6	-
1	0.5	0.2	0.5	-	0.1	0.5	-
1.5	0.05	0.2	0.4	-	0.1	0.1	-
1.5	0.25	0.2	0.5	-	0.1	0.5	-
1.5	0.5	0.1	0.1	-	0.1	0.1	-

U: the number of unsolved instances

Table 5.8: Results for 45-job instances.

		New			*Old*		
α	β	*Avg.*	*Max.*	*U*	*Avg.*	*Max.*	*U*
0	0.05	0.1	0.2	-	0.3	0.8	-
0	0.25	1.5	3.7	-	61.3	152.0	-
0	0.5	14.6	37.9	-	1570.6	3600	1
0.5	0.05	1.8	5.1	-	13.7	67.3	-
0.5	0.25	25.9	54.8	-	1827.7	3600	3
0.5	0.5	1549.1	3600	2	3600	3600	10
1	0.05	1.8	4.1	-	2.0	6.1	-
1	0.25	1.0	3.3	-	0.7	3.1	-
1	0.5	0.2	0.3	-	0.1	1.0	-
1.5	0.05	0.3	0.4	-	0.1	0.2	-
1.5	0.25	0.2	0.5	-	0.1	0.3	-
1.5	0.5	0.1	0.2	-	0.1	0.1	-

U: the number of unsolved instances

Table 5.9: Results for 50-job instances.

		New			*Old*		
α	β	*Avg.*	*Max.*	*U*	*Avg.*	*Max.*	*U*
0	0.05	0.1	0.2	-	0.3	0.6	-
0	0.25	2.9	9.1	-	101.7	330.4	-
0	0.5	56.6	197.9	-	1522.1	3227.9	-
0.5	0.05	4.5	18.1	-	19.7	67.4	-
0.5	0.25	73.7	142.1	-	3006.8	3600	6
0.5	0.5	2466.6	3600	5	3600	3600	10
1	0.05	1.9	4.5	-	2.4	6.5	-
1	0.25	2.1	13.7	-	2.6	21.2	-
1	0.5	0.2	0.4	-	0.1	0.1	-
1.5	0.05	0.5	0.7	-	0.1	0.3	-
1.5	0.25	0.3	0.4	-	0.1	0.1	-
1.5	0.5	0.2	0.3	-	0.1	0.1	-

U: the number of unsolved instances

Table 5.10: Results for 60-job instances.

		New			*Old*		
α	β	*Avg.*	*Max.*	*U*	*Avg.*	*Max.*	*U*
0	0.05	0.4	0.7	-	1.6	3.5	-
0	0.25	18.7	51.1	-	834.3	2189.8	-
0	0.5	570.2	1676.1	-	3600	3600	10
0.5	0.05	38.8	112.1	-	564.1	3600	1
0.5	0.25	1380.0	3457.6	-	3600	3600	10
0.5	0.5	3388.5	3600	9	3600	3600	10
1	0.05	29.1	148.9	-	50.2	330.2	-
1	0.25	59.6	564.0	-	93.8	798.6	-
1	0.5	0.6	0.9	-	0.1	0.3	-
1.5	0.05	1.6	2.3	-	0.5	1.6	-
1.5	0.25	0.7	1.0	-	0.1	0.2	-
1.5	0.5	0.5	0.6	-	0.1	0.1	-

U: the number of unsolved instances

Chapter 6

Conclusions and Future Work

In this paper, we introduce an efficient branch and bound algorithm for a strongly NP-Hard problem. First, by utilizing the multiplier adjustment method, we get a new lower bounding scheme. In particular, we show that the new lower bounding scheme gives better lower bounds than the lower bounding scheme from the weighted completion time problem in most cases. Secondly, based on a modified local search heuristic, we get a better upper bounding scheme at the root node of the search tree. In Section 4.2, we suggest a data structure for an important dominance rule.

Numerical results show that utilization of the new lower bounding scheme significantly decreases the computational time for difficult test instances. Working together with the lower bounding scheme from the weighted completion problem, the new lower bounding scheme enables the branch and bound algorithm to solve the hardest test instances with up to 40 jobs in one hour time limit. Compared with the results of the old solver from Jouglet et al. [21], our numerical results of the new solver show that the three new techniques decrease computational time for most test instances and require less computational resources.

There are a few issues that need further exploration for the single machine scheduling problem. First, our enumerative algorithm is still a kind of branch

and bound method. It seems that the dynamic programming approach may be an attractive alternative, see [32].

Secondly, we point out that both the new lower bounding scheme and the lower bounding scheme from the total weighted completion time problem can not be directly applied to other single machine scheduling problems.

We note that in [32], the authors propose an efficient enumerative algorithm for the single machine total weighted tardiness problem *without* release dates. This algorithm is based on the *successive sublimation dynamic programming* [32] method with some improvements. They use *state-space relaxation* [32] to get a lower bound. The lower bounding scheme is improved by the dominance of two adjacent jobs. Combined with other improvements, their algorithm can handle test instances with up to 300 jobs. Our next step is to explore these ideas and test for the problem with release dates.

Finally, the ideas of the three new techniques can be applied to other single machine scheduling problems with release dates. It will be interesting to test and compare the three new techniques with other techniques for various problems.

Bibliography

[1] T.S. Abdul-Razaq, C.N. Potts and L.N. Van Wassenhove. A survey of algorithms for the single machine total weighted tardiness scheduling problem. *Discrete Applied Mathematics*, 26:235-253, 1990.

[2] M.S. Akturk and D. Ozdemir. An exact approach to minimizing total weighted tardiness with release dates. *IIE Transactions*, 32:1091-1101, 2000.

[3] M.S. Akturk and D. Ozdemir. A new dominance rule to minimize total weighted tardiness with unequal release dates. *European Journal of Operational Research*, 135:394-412, 2001.

[4] K.R. Baker. *Introduction to Sequencing and Scheduling.* JohnWiley and Sons, 1974.

[5] H. Belouadah, M.E. Posner, and C.N. Potts. Scheduling with release dates on a single machine to minimize total weighted completion time. *Discrete Applied Mathematics*, 36:213-231, 1992.

[6] L. Bianco and S. Ricciardelli. Scheduling of a single machine to minimize total weighted completion time subject to release dates. *Naval Research Logistics*, 29:151-167, 1982.

[7] J. Carlier. Ordonnancements à contraintes disjonctives. *RAIRO*, 12:333-351, 1978.

[8] C. Chu. A Branch and Bound Algorithm to Minimize Total Flow Time with Unequal Release Dates. *Naval Research Logistics*, 39:859-875, 1991.

[9] C. Chu. A Branch and Bound Algorithm to Minimize Total Tardiness with Different Release Dates. *Naval Research Logistics*, 39:265-283, 1992.

[10] C. Chu. Efficient Heuristics to Minimize Total Flow Time with Release Dates. *Operations Research Letters*, 12:321-330, 1992.

[11] C. Chu and M.C. Portmann. Some new efficient methods to solve the $1|r_i|\sum T_i$ scheduling problem. *European Journal of Operational Research*, 58:404-413, 1991.

[12] R.K. Congram, C.N. Potts and S.L. van de Velde. An iterated dynasearch algorithm for the single machine total weighted tardiness scheduling problem. *INFORMS Journal of Computing*, 14:52-67, 2002.

[13] R.W. Conway, W.C.Maxwell and L.W. Miller. *Theory of Scheduling*. AddisonWesley, Reading, MA, 1967.

[14] T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein. *Red-Black Trees*. Chapter 13 in Introduction to Algorithms, Second Edition, T.H. Cormen et al. The MIT Press, Cambridge, MA, USA, 2001.

[15] J. Du and J.Y.T. Leung. Minimizing total tardiness on one processor is np-hard. *Mathematics of Operations Research*, 15:483-495, 1990.

[16] M.L. Fisher. The Lagrangean Relaxation Method for Solving Integer Programming Problems. *Management Science*, 27:1-18, 1981.

[17] A.M. Geoffrion. Lagrangean Relaxation for Integer Programming. *Math. Program. Study* 2:82-114, 1974.

[18] L. Guibas and R. Sedgewick: A Dichromatic Framework for Balanced Trees. *FOCS* 1978: 8-21.

[19] A. Hariri and C.N. Potts. An Algorithm for Single Machine Sequencing with Release Dates to Minimize Total Weighted Completion Time. *Discrete Applied Mathematics*, 5:99-109, 1983.

[20] A. Jouglet. Ordonnancer une Machine pour Minimiser la Somme des Coûts - *The One Machine Total Cost Sequencing Problem.* PhD thesis, Université de Technologie de Compiègne, Compiègne, France, 2002.

[21] A. Jouglet, P. Baptiste and J. Carlier. *Branch-and-Bound Algorithms for Total Weighted Tardiness.* Chapter 13 in Handbook of Scheduling: Algorithms, Models, and Performance Analysis, Joseph Y-T. Leung, CRC Press, Boca Raton, FL, USA, 2004.

[22] A.H. Land and A.G. Doig, An Automatic Method for Solving Discrete Programming Problems. *Econometrica*, 28:497-520, 1960.

[23] E.L. Lawler. A pseudo-polynomial algorithm for sequencing jobs to minimize total tardiness. *Annals of Discrete Mathematics*, 1:331-342, 1977.

[24] E.L. Lawler. Sequencing jobs to minimize total weighted completion time subject to precedence constraints. *Ann. Discrete Math.* 2:75-90, 1978.

[25] J.K. Lenstra, A.H.G. Rinnooy Kan and P. Brucker, Complexity of machine scheduling problems. *Ann. Discrete Math.* 1:343-362, 1977.

[26] M. Pinedo *Deterministic and Stochastic Dynamic Programming.* Appendix B in Scheduling: Theory, Algorithms, and Systems, M. Pinedo, Prentice-Hall, 2008.

[27] M.E. Posner. Minimizing weighted completion times with deadlines. *Oper. Res.* 33:562-574, 1985.

[28] C.N. Potts and S.L. van de Velde. *Dynasearch - iterative local improvement by dynamic programming. Part I: the traveling salesman problem.* Technical report, University of Twente, 1995.

[29] C.N. Potts, L.N. Van Wassenhove. A branch and bound algorithm for the total weighted tardiness problem. *Operations Research* 33:363-377, 1985.

[30] L. Schrage and K.R. Baker. Dynamic Programming Solution of Sequencing Problems with Precedence Constraints. *Opns. Res.* 26, 444-449, 1978.

[31] W.E. Smith. Various Optimizers for Single-Stage Production. *Naval Res. Logist. Quart.* 3:59-66, 1956.

[32] S. Tanaka, S. Fujikuma and M. Araki, An Exact Algorithm for Single Machine Scheduling without Machine Idle Time, *Journal of Scheduling.* DOI: 10.1007/s10951-008-0093-5, 2008

[33] L.N. Van Vassenhove. *Special-Purpose Algorithms for One-Machine Sequencing Problems with Single and Composite Objectives.* Ph.D. thesis, Katholieke Universiteit Leuven. 1979.

[34] A.P.J. Vepsalainen and T.E. Morton. Priority Rules for Job Shops with Weighted Tardiness Costs. *Management Science.* 33:1035-1047, 1987.

[35] G. Verfaillie and T. Schiex. Solution reuse in dynamic constraint satisfaction problems. *AAAI94*, Seattle,WA, U.S., 307C312, 1994.