CHAPTER IV
LOWER BOUNDS

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

LIST OF ACRONYMS

A(s)	The arc set coming from source node s
ASB	Adjusted Set Bound
ASSB	Adjusted Subset Bound
DHB	Dual Heuristic Bound
HFS	Hybrid Flowshop
HLB	HFS Lower Bound
HSSB	Heuristic for the SSB
JB	Job Bound
LB(s)	Lower Bound at stage s
LP	Linear Program
MSS	The maximum optimal Makespan over all Single Stage subproblems
PB	Preemptive Bound
PPB	Partially Preemptive Bound
RB	Rest Bound
SB	Set Bound
SRB	Subset Rest Bound
SSB	Subset Bound

CHAPTER IV
LOWER BOUNDS

Abstract

This chapter studies lower bounds on the makespan minimization of the hybrid flowshop scheduling problem. All lower bounds are based upon the single stage subproblem lower bounds. We recall some of known lower bounds and set the outline of the research conducted in the development of new and tighter lower bounds.

A lower bound for the single stage subproblem is the Preemptive Bound (PB), which is the optimal solution for the preemptive case. PB is obtained in polynomial time, viz. $O(p_{max}n^3)$ where n is the number of jobs and p_{max} is the largest processing time, but the computation complexity is too high for use in a branch and bound framework. A less time consuming but less tight lower bound is the subset bound (SSB) which can be computed in $O(n^2)$. This chapter introduces two new tighter lower bounds and a heuristic for SSB. The first lower bound is a Dual Heuristic based Bound (DHB) induced by a time indexed Linear Program (LP) formulation where the objective is to build a non-preemptive optimal schedule. The second lower bound is obtained by constraining the preemptive assignment of jobs. It is also based on a time indexed LP formulation which produces a partially preemptive schedule (PPB). As a matter of fact, this time-indexed formulation constrains the preemptive assignment of the jobs which makes PPB tighter than PB, but its running time complexity is higher. The computation of PB and PPB is obtained by bisection using transportation problems. "Preflow-Push" algorithms are used to solve such network problems. An improved "Preflow-Push" algorithm which runs faster than the original version is proposed. The third lower bound is a Heuristic for SSB (HSSB) which can be calculated in a time complexity of O(nM) where M is the number of machines. All lower bounds are tested on several single stage configurations using several data ranges so as to study their behaviors and to select the best-suited ones to be used in a branch and bound algorithm.

1. INTRODUCTION

This chapter studies lower bounds on the makespan minimization of the hybrid flowshop (HFS) scheduling problem. A first lower bound that comes to mind when minimizing the makespan in the non-preemptive case is the preemptive bound obtained by allowing job preemption. However, finding the minimum makespan of a two-stage HFS scheduling problem when preemption is allowed is NP-hard. See proof in Chapter II of Hoogeveen et al., (1996). A lower bound can then be obtained by relaxing the HFS problem into a single stage subproblem with release dates and tails (see Chapter II, Section 5). In this case a global HFS lower bound (HLB) can be obtained by retaining the maximum optimal Makespan over all Single Stage relaxation or subproblems s=1, …, K, say $MSS^*(s)$. Since the complexity of finding the optimal makespan of the single stage s subproblem can be very high (see Carlier 1987), any lower bound LB(s) on that

makespan is also a lower bound on the makespan of the HFS. A lower bound on the makespan of the HFS scheduling problem can then be computed as follows:

$$HLB1 = \max_{s\in\{1,...,K\}} MSS^*(s) \qquad (4.1)$$

$$HLB2 = \max_{s\in\{1,...,K\}} LB(s) \qquad (4.2)$$

As shown in equation (4.2), and in order to come up with an HFS lower bound, we need to take the maximum lower bound over all stages and, consequently, need to compute K single stage lower bounds. If we decide to compute an HFS tighter and more time consuming lower bound, we do not need, in some cases, to compute the maximum over all single stage lower bounds. Let UB(s) and LB(s) be, respectively, an upper bound and lower bound on the makespan of the single stage s subproblem. Suppose that LB(s) is obtained with a less time consuming algorithm, e.g. the set bound computed in O(n) (see Section 2). Let LB^* be the maximum lower bound over all the K stages, i.e. $LB^* = \max_{s\in\{1,...,K\}}$ LB(s). For any stage s, there is no need to compute a tighter lower bound when UB(s)≤LB*. In the following example there is no need to improve the lower bound of the single stage s = 1 and s = 3. The lower bounds at these two stages, at most, would be equal to their corresponding upper bounds, which are equal to and smaller than LB(2), respectively.

Example 4.1: LB* = 13

stage	1	2	3
UB	13	15	12
LB	10	13	8

In the remainder of this chapter we present only lower bounds for the single stage subproblem, which will be used for computing lower bounds on the makespan of the HFS schedule using equation (4.2). We believe that lower bounds based on more than one stage are unlikely to exist. Such lower bounds seem much harder to develop. Actually, by considering release dates and tails, the single stage s subproblem does take into account the information on the other stages t≠s. However, this information comes from the relaxation of the number of machines at the other stages t≠s, see Chapter II, Section 4.

In our notation of the single stage subproblem data, we will ignore the subscript s and assume that these data are always related to some stage s with respect to the HFS corresponding problem. We also assume that integral release dates r_i and tails q_i are available for all jobs.

We now recall the studied single stage subproblem.

Problem definition. We need to schedule a set of jobs I={1, …, n} on M identical parallel machines. Each job has a processing time p_j and may have a release date r_j, and a tail q_j. The objective is to minimize the makespan C_{max} so that each job cannot start before its earliest start date r_j+1 and that its processing should be completed before time C_{max}-q_j, i.e. before its due date $d_j = C_{max} - q_j$. Job preemption and job splitting are not allowed. All data of the problem (p_j, r_j and q_j, j∈I) are integer. For the release date and

tail definitions as well as for a detail description of the single stage subproblem see Chapter II.

NEW LOWER BOUND DEVELOPMENTS

Minimization of the makespan of the single stage subproblem in the non-preemptive case can be formulated as a minimization Integer Linear Program ($\min^{ILP}$). From such formulation, i.e. ($\min^{ILP}$), where the makespan is *minimized* using integer variables, different *lower bounds* can be derived. A first lower bound can be computed by relaxing the variables integrality and obtaining a pure minimization Linear Program ($\min^{LP}$) which can be solved by the Simplex Method. However, for large problems, solving ($\min^{LP}$) might also be time consuming. Therefore, for computing a lower bound on $\min^{LP}$ and consequently on $\min^{ILP}$ since $\min^{LP} \leq \min^{ILP}$, one can approximate the ($\min^{LP}$) objective function using a heuristic for the maximization of its dual formulation. According to the weak duality theorem, any objective function value, say DHB, of a maximization dual formulation ($\max^{DLP}$), is a lower bound on the corresponding optimal value for a minimization problem ($\min^{LP}$), i.e. DHB $\leq \max^{DLP} = \min^{LP}$. Figure 4.1(a) depicts graphically the relation between $\min^{ILP}$, $\min^{LP}$ and DHB. In Section 3 we present such a lower bound (DHB) for the single stage subproblem.

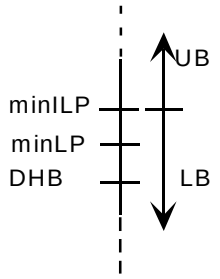


Figure 4.1(a) :
Relation between $\min^{ILP}$, $\min^{LP}$ and DHB

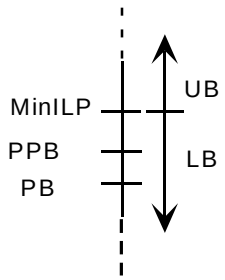


Figure 4.1(b) :
Relation between $\min^{ILP}$, PPB and PB

More specifically for scheduling problems, and since we are seeking the minimum makespan of a non-preemptive schedule, one can compute a lower bound on that makespan by allowing job preemption. The lower bound is obtained by building an *optimal* preemptive schedule, also called Preemptive Bound (PB). For the single stage subproblem this lower bound can be computed by bisection using a maximum flow algorithm.

Moreover, one can also constrain job preemption and obtain a tighter Partially Preemptive Bound (PPB) where jobs starting times are more constrained than in the case of PB. Figure 4.1(b) depicts graphically the relation between $\min^{ILP}$, PPB and PB. Note that in Figures 4.1(a) and 4.1(b) we do not depict the relation between $\min^{LP}$ (and DHB) with PB and PPB. These theoretical relations are unknown to us.

The remainder of this chapter is organized as follows. Section **2** recalls some of known lower bounds for the single stage subproblem. Section **3** presents a time indexed formulation for the single stage makespan minimization problem, whose relaxation allows building several lower bounds. Section **4** introduces a Dual Heuristic based Bound

(DHB). The latter is derived from the time indexed ILP formulation where the objective is to build a non-preemptive optimal schedule. Section **5** recalls the pure Preemptive lower Bound (PB). Section **6** presents how to build a Partially Preemptive Bound (PPB). This lower bound is also derived from a time indexed ILP relaxation formulation, which is a transportation problem. The main characteristic of this formulation is that the preemptive assignment of jobs is constrained and therefore the lower bound is tighter than PB. PB and PPB are obtained by solving a set of "*maximum-flow/minimum-cut*" problems. The latter problem simply checks whether a certain trial value δ is feasible in the sense that it allows building a preemptive or a partially preemptive schedule. Section **7** recalls the "*Preflow-push"* algorithm we decided to use for solving the network problem, and presents some implementation improvements. The so-called subset bound can be computed in $O(n^2)$. Section **8** presents a heuristic for the latter (HSSB) computed in a time complexity of O(nM), where n is the number of jobs and M is the number of machines. Section **9** presents numerical tests comparing all lower bounds presented in this chapter on different single stage configurations. Section 10 concludes this chapter.

2. LOWER BOUNDS FOR THE SINGLE STAGE SUBPROBLEM

A set of lower bounds of various quality and complexity regarding the problem of minimizing the makespan of the single stage subproblem has been proposed in the literature. In Vandevelde et al., (1995) the interested reader can find an extensive development of these bounds for which we hereafter give a brief description illustrated with example 4.2.

Example 4.2: I = {1, 2, 3, 4, 5, 6}, M = 2

j	r_j	p_j	q_j	$r_j + p_j + q_j$
1	3	6	3	12
2	1	1	1	3
3	1	2	1	4
4	3	2	3	8
5	4	2	2	8
6	3	2	3	8

The Job Bound (JB) (4.3) of complexity *O(n)* relaxes the number of machines by assuming that the number of machines is as large as the number of jobs. In such relaxed problem the makespan is given by $\max_{j \in I} (r_j+p_j+q_j)$. In Example 4.2, JB(I) = 12 and is given by job 1.

$$JB(I) = \max_{j \in I} (r_j + p_j + q_j) \qquad (4.3)$$

The Set Bound (SB) (4.4) comes from the observation that the makespan must contain at least the M smallest release dates and tails as well as the total processing time at stage s. At best, this time can be evenly distributed among the M parallel machines. SB is computed in O(n). In Example 4.2, SB(I) = $\lceil (1 + 1 + 1 + 1+ 15) / 2 \rceil = 10$.

$$SB(I)= \lceil (r_{[1]} + r_{[2]}+\ldots+ r_{[M]} + q_{[1]} + q_{[2]} +\ldots + q_{[M]} + \sum_{j \in I} p_j) / M \rceil \qquad (4.4)$$

where $r_{[1]}$, $r_{[2]}$, ..., $r_{[M]}$ are the M smallest r_j over $j \in I$
$q_{[1]}$, $q_{[2]}$, ..., $q_{[M]}$ are the M smallest q_j over $j \in I$

The Subset Bound (SSB) (4.5) relaxes the problem by considering only a subset of jobs and uses SB on this subset for the computation of the lower bound. Vandevelde et al., (1995) have shown that SSB can be calculated in $O(n^2)$ without enumerating all subsets of I. They first turn equation (4.5) defining SSB(I) into a *"max-min"* integer linear program, which is then reformulated as equation (4.5').

$$SSB(I) = \max_{V \subset I, |V| > M} SB(V) \qquad (4.5)$$

$$M*SSB(I) = \max\{ \max_{J \subseteq I, |J| = M} \sum_{j \in I} (p_j + r_j + q_j), \max_{\substack{u = r_1, r_2, \ldots, r_n \\ v = q_1, q_2, \ldots, q_n \\ |J^+(u,v)| > M}} \left(M(u+v) + \sum_{j \in J^+(u,v)} (p_j - (u - r_j)^+ - (v - q_j)^+) \right) \}$$

$$\text{where } J^+(u,v) = \{ j \in I \mid p_j - (u - r_j)^+ - (v - q_j)^+ > 0 \} \qquad (4.5')$$

A straightforward implementation of (4.5') would result in an $O(n^3)$. First, they establish, in order O(nlog n), several ordering of set I in non decreasing values of r_j, q_j, p_j, $p_j + q_j$ and $p_j + r_j + q_j$, respectively. Then, by setting up for each u, in order O(n), auxiliary arrays A_1 and A_2 giving, for each job j, the first index *k* for which $p_j + q_j - q_{[k]} \leq 0$ ($q_{[k]}$ stands for the k^{th} smallest tail), and the first index *l*, for which $p_j + r_j + q_j - u - q_{[l]} \leq 0$, respectively, it is possible to compute equation (4.5') in $O(n^2)$. The complete demonstration can be found in Vandevelde et al., (1995).

In Example 4.2, $SSB(I) = SB(I \setminus \{2,3\}) = \lceil (3 + 3 + 3 + 2 + 12) / 2 \rceil = 12$

The Adjusted Set Bound (ASB) comes from the fact that in any non-preemptive schedule there are two options:

- Some machine processes job j and the other machines process all other jobs
- Job j is not the first and the last job to be processed on a machine

In the first case two lower bounds can be used, one on the machine with the single job j and one on the other machines and jobs. In the latter case SB can be strengthened by adding the conditions that at most one r_j and q_j appear in the bound. To use this observation Vandevelde et al., (1995) introduce the notation *SB(A,B,c)*, where A is the set of jobs for which they calculate the bound (a set bound), B is the set of jobs of which not both head and tail appear in the bound, and c is the number of machines on which the jobs in A have to be processed. As long as there is a job whose both head and tail appear in the bound the bound SB can be improved. They define the *Adjusted Set Bound ASB(A,B,c)* as the optimal value found using this procedure recursively:

ASB(A,B,c)

If there is a job j of which both r_j and q_j appear in SB(A,B,c)
Then ASB(A,B,c) = min {max{$r_j+p_j+q_j$, ASB(A\{j},B,c-1)}, ASB(A,B∪{j},c) }
Else ASB(A,B,c) = SB(A,B,c)

A lower bound on the single stage subproblem is then given by *ASB(I,∅,M)*, or ASB for short. ASB can be computed in O(*nlogn***2*^{2M}).

In Example 4.2, both r_2 and q_2 appear in SB (used in the computation of SB). Since $r_2+p_2+q_2=3$ and SB(I) = 10, either r_2 or q_2 appears in the optimal makespan but not both. So we can compute ASB(I, I∪{2},2) = $\lceil (3 + 1 + 1 + 1 + 15) / 2 \rceil = 11$ where r_2 does not appear in the computation of SB. In this example $r_2 = 1$ has been replaced by $r_1=3$ in SB.

The Adjusted Subset Bound (ASSB). Vandevelde et al., (1995) also claim that computing ASB for every subset of I and taking the maximum would yield a strong lower bound. Doing so is time consuming because it requires enumerating all subsets of I. They propose to compute the *ASSB* only for the subset I* found by SSB.

$$\text{ASSB(I)} = \text{ASB(I*,}\varnothing\text{,M)} \text{ where SB(I*)} = \max_{V \subset I, |V|>M} \text{SB(V)}.$$

The Rest Bound (RB) can be used when the number of machines does not divide the number of jobs. Define M* = n mod M and n* = M*$\lceil$n/M$\rceil$. The M* machines with the most jobs together process at least n* jobs. RB is defined by:

$$\text{RB(I)} = \lceil (r_{[1]} + r_{[2]} + \ldots + r_{[M^*]} + q_{[1]} + q_{[2]} + \ldots + q_{[M^*]} + p_{[1]} + p_{[2]} + \ldots + p_{[n^*]}) / M^* \rceil \quad (4.6)$$

where $r_{[1]}, r_{[2]}, \ldots, r_{[M^*]}$ are the M* smallest r_j over $j \in I$

$q_{[1]}, q_{[2]}, \ldots, q_{[M^*]}$ are the M* smallest q_j over $j \in I$

$p_{[1]}, p_{[2]}, \ldots, p_{[n^*]}$ are the n* smallest p_j over $j \in I$

The Subset Rest Bound (SRB) computes RB for small subsets of jobs with large processing times and is defined as follows:

$$\text{SRB} = \max_{k=1,\ldots,n} \text{RB}(I_k) \quad (4.7)$$

where $I_k = \{j | p_j \geq p_{[k]}, j \in I\}$. This bound can be computed in O(n^2) time.

Figure 4.2 presented in Vandevelde et al., (1995) recalls the *theoretical* relations between these bounds, where *Makespan** is the optimal makespan of the single stage subproblem and where A→ B if A is dominated by B, i.e. A gives at most the same value as B.

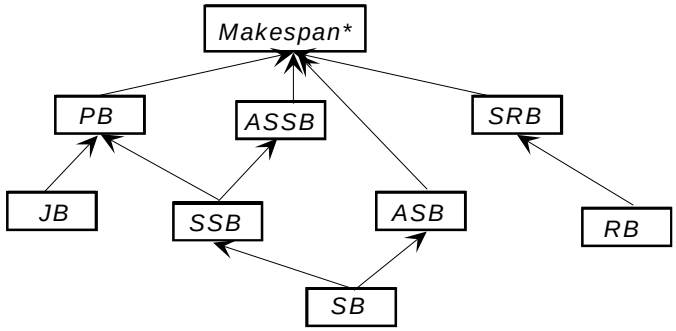


Figure 4.2: *Theoretical* relation between bounds for the single stage subproblem (Vandevelde et al., (1995))

We have tested all these bounds on several single stage subproblem configurations and *for nearly all tested problems (see later Table 4.6 in Section 9)* the following conclusions can be made:

- SRB is always dominated by SB and is much more time consuming than SB
- ASB gives always the same value as SB and is much more time consuming
- ASSB gives always the same value as SSB and is much more time consuming

Furthermore, Vandevelde et al., (1995) claim that PB - max(JB, SSB) ≤ p_{max}*(M-1)/M where p_{max} is the maximum processing time in set I. Also, from their numerical tests they observed that for all their test instances, max(JB, ⌈SSB⌉) = PB, which was also confirmed by our numerical tests, see Section 9.

Therefore we decided to concentrate on finding lower bounds tighter than PB or SSB and to improve the running time complexity of the SSB.

Three new lower bounds are introduced in this chapter: a Dual Heuristic Bound (DHB), a Partially Preemptive Bound (PPB) as well as a heuristic for the subset bound (HSSB). We first present a mathematical formulation for finding the minimum makespan in the non-preemptive case from which releaxtion DHB and PPB are desinged.

3. MINIMIZING THE MAKESPAN IN THE NON-PREEMPTIVE CASE

We hereafter present a mathematical formulation for finding the minimum makespan in the non-preemptive case. This formulation, given a makespan trial δ and the number of machines M, maximizes the total processing times P(δ). The solution found is considered feasible only if $P(\delta) \geq \sum_{j \in I} p_j$. Given a lower and upper bound (LB and UB) on the optimal makespan δ*, bisection on the interval [LB, UB] allows finding the optimal value δ^*, without checking all values δ between LB and UB. The smallest value of δ∈[LB, UB] that produces a feasible schedule is the optimal solution, i.e. δ*. The optimal feasible solution δ* is equal to (δ+1) where δ produces an infeasible non-preemptive schedule and (δ+1) produces a feasible one. The optimal makespan δ* is then defined by: δ*=Min δ s.t. $P(\delta) \geq \sum_{j \in I} p_j$

This formulation will be relaxed to design a Dual Heuristic Bound (DHB), as well as the preemptive and the partially preemptive bounds, i.e. PB and PPB. Figure 4.3 gives a schematic view how these bounds will be induced from the non-preemptive formulation.

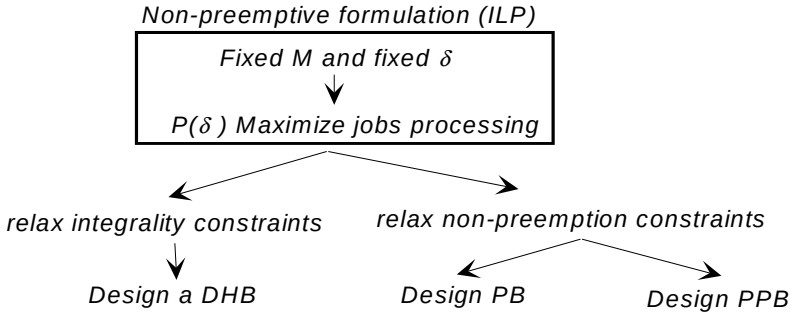


Figure 4.3: Schematic view of the designed bounds

MAXIMIZING JOBS PROCESSING

Given a trial value δ, the following formulation aims at producing a non-preemptive schedule of the single stage subproblem. This problem is divided into two subproblems. Problem (P(δ)), given some trial makespan δ, maximizes the total processing times. Using subproblem (P(δ)), problem (U) finds the minimum makespan δ* such that P(δ*), i.e. the total processing time permitted by δ*, is at least equal to the total processing time of the jobs in set I.

Let z_{jt} be a binary variable equal to 1 if job j starts processing in period t and 0 otherwise. This variable also means that if $z_{jt} = 1$ the machine to which job j is assigned to is busy from time t until time $(t + p_j - 1)$. Figure 4.4 shows the possible starting periods of a job j as well as the periods when the machine to which this job is assigned to is busy processing it when $z_{jt}=1$.

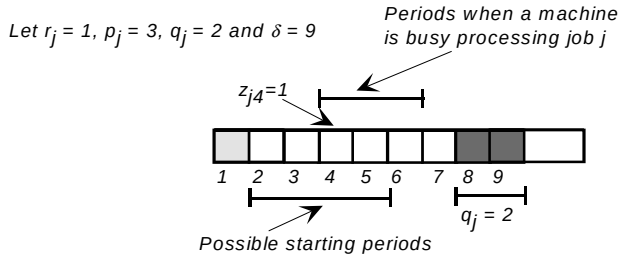


Figure 4.4: Possible starting periods of a job j and periods when a machine is busy processing that job

This time indexed formulation has been introduced by Sousa and Wolsey (1992) for the single machine problem. See Sousa and Wolsey (1992), and Sousa (1989) for a detailed study of such a formulation for solving the one-machine scheduling problem.

$$(P(\delta)) \quad P(\delta) = \max \sum_{j\in I} \sum_{t=r_j+1}^{\delta-q_j-p_j+1} z_{jt} * p_j \qquad \text{s.t.} \tag{4.8.0}$$

$$\sum_{t=r_j+1}^{\delta-q_j-p_j+1} z_{jt} \le 1 \qquad \forall\, j\in I \tag{4.8.1}$$

$$\sum_{j\in I} \sum_{\substack{s=\max(r_j,\,t-p_j+1) \\ \text{with } s\le \delta-p_j-q_j+1}}^{t} z_{js} \le M \text{ for } t= \min_{j\in I}(r_j)+1, \ldots, \delta- \min_{j\in I}(q_j + p_j) +1 \tag{4.8.2}$$

$$z_{jt} \in \{0,1\} \qquad \forall\, j\in I,\ t = r_j +1, \ldots, \delta-p_j-q_j+1 \tag{4.8.3}$$

$$(U) \quad \delta^* = \min \delta \qquad \text{s.t.} \tag{4.8.4}$$

$$P(\delta) = \sum_{j\in I} p_j \tag{4.8.5}$$

Constraint set (4.8.1) states that each job starts at most once within its possible starting time interval, i.e. $[r_j+1, \delta-p_j-q_j+1]$. Job j has to start at time $t = \delta - p_j - q_j + 1$, at the latest, so that its completion time does not exceed the given limit of time δ. Constraints (4.8.1) have been relaxed allowing some jobs not to start if necessary, because δ might be not large enough to process all jobs. Constraints set (4.8.2) states that the number of machines M is larger than or equal to the number of busy machines at any time t. If job j

starts processing, i.e. $z_{js}=1$, within the time interval $[t-p_j+1, t]$, i.e. $\sum_{s=t-p_j+1}^{t} z_{js}=1$, in the non-preemptive case, this job is still being processed at time t. In that case, within this time interval $[t-p_j+1,t]$ one machine is assigned to that job. Hence the left hand side of (4.8.2) represents the number of busy machines at time t.

Formulation P(δ) maximizes the processing of jobs (we process as many jobs as possible) so that the makespan does not exceed the imposed limit of time δ. It tackles only a subproblem of the minimization of the makespan. It answers the following question: does a given value of δ allow processing all jobs? In other words, is $P(\delta) = \sum_{j\in I} p_j$?

Knowing whether δ permits processing all jobs or not, the second problem is to find the smallest value of δ allowing the scheduling of all jobs. Formulation (U), find the minimum δ such that $P(\delta)= \sum_{j\in I} p_j$.

Solving the single stage subproblem with reasonable sizes using formulation P(δ) is out of the capability of any existent optimization software. Indeed P(δ) is a time indexed Integer Linear Program with a large number of variables, depending on the number of jobs and the length of the trial makespan value. Even solving the linear relaxation of P(δ) is time consuming so as to come up with a lower bound on δ*. We therefore suggest to develop a dual heuristic for such linear relaxation.

4. DUAL HEURISTIC BOUND (DHB)

Based on formulation P(δ), we present a dual based heuristic that allows finding a lower bound on the makespan δ*. We first need to relax the integrality constrains (4.8.3), i.e. $1\geq z_{jt}\geq 0$, $\forall j\in I$ and for all t, so as to obtain a Linear Program $P(\delta)^{LP}$. Then based on the dual formulation of $P(\delta)^{LP}$ we design a Dual Heuristic Bound DHB.

In Figure 4.1(a) we presented the relation between an integer linear program ($\min^{ILP}$), its linear programming relaxation ($\min^{LP}$) and any dual based heuristic. Conversly, in our case $P(\delta)\leq P(\delta)^{LP}\leq DHB$ since we are tackling a maximization problem.

DUAL FORMULATION

Let α_j and β_t be the dual variables associated with constraint sets (4.8.1) and (4.8.2), respectively. The dual formulation of the relaxed problem $P(\delta)^{LP}$, i.e. problem P(δ) where constraints set (4.8.3) is replaced by set $1\geq z_{jt}\geq 0$ for all $j\in I$ and for all t, is as follows:

$$(\mathbf{D}(\delta))\quad W^* = \min W = \sum_{j\in I}\alpha_j + M^* \sum_{\substack{t=\min_{j\in I}(r_j)+1}}^{\delta-\min_{j\in I}(q_j-p_j)+1} \beta_t \qquad \text{s.t.} \qquad (4.9.1)$$

$$\alpha_j + \sum_{s=t}^{t+p_j-1}\beta_s \geq p_j \qquad \forall j,\ \forall\ t = r_j+1,\ldots,\ \delta - q_j - p_j + 1 \qquad (4.9.2)$$

$$\alpha_j,\ \beta_t \geq 0 \qquad (4.9.3)$$

4.1 DESIGN OF THE DUAL HEURISTIC BOUND

To design a dual heuristic we proceed as follows:

- With a given value δ we first build a feasible solution for problem (D(δ)) (where all constraints are satisfied) and compute its objective function value W. W is a upper bound on D(δ).
- We then go through an improvement process where at each iteration k we try to reduce the value of W, trying to reach the optimal value of W*, while maintaining the feasibility for problem (D(δ)), until no improvement could be achieved.

Figure 4.5 shows the relation between the optimal values P(δ)*, $P(\delta)^{LP}$*, W* and any feasible value W of problem (D(δ)). Note that $P(\delta)^{LP}$* = W* because problem (D(δ)) is the dual of problem (P(δ)).

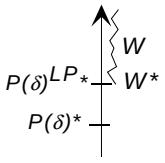


Figure 4.5: Relation between P(δ)*, $P(\delta)^{LP}$*, W* and W

With a trial value δ and each time we come up with an improved W, one of the following two situations might occur:

1. The newly obtained $W < S^{p_j}$, we then conclude that the checked δ value is too small to allow processing all jobs. δ can then be increased to allow more processing of jobs and δ+1 is certainly a lower bound on δ*.
2. The new value $W \geq S^{p_j}$, we can still suppose that δ is large enough to allow processing all jobs and we try then once again to reduce W. The heuristic stops when $W \geq S^{p_j}$ and we no longer succeed to reduce W.

A lower bound on the makespan δ* is δ' if δ'-1 produces situation 1, i.e. δ'-1 is too small to allow processing all jobs, and δ' produces situation 2, i.e. δ' is large enough to allow processing all jobs and W can no longer be decreased.

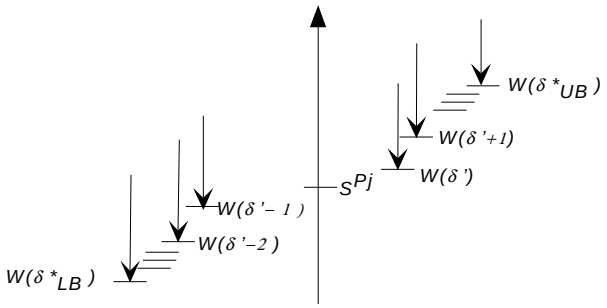


Figure 4.6: A schematic view of different values of W obtained by different values of δ

In order to find a lower bound on the optimal δ* we need to carry out bisection on a certain interval on δ*, [δ^*_{LB}, δ^*_{UB}], see later Figure 4.8 in this section. Figure 4.6 shows a

schematic view of different values of W obtained by different values of δ, where W(δ) is the value obtained by DHB using δ. DHB, here, stops with δ' as a lower bound on δ* since $W(\delta'-1) < S^{p_j}$ and $W(\delta') > S^{p_j}$. Note also that depending on the effectiveness of the dual heuristic we can stop with δ' while in the optimal value of the dual problem $W^*(\delta') \leq S^{p_j}$, which means that there is a better bound than the one found by DHB, i.e. δ'.

Let

- $C_t = \sum_{j \in I} \sum_{s=r_j+1}^{\delta-q_j-p_j+1} c_{j,s}^t$ where $c_{j,s}^t$ is the coefficient of β_t in constraint (j, s) in constraints set (4.9.2). In other words, C_t is the number of constraints in the set (4.9.2) where variable β_t appears with a positive (=1) coefficient. See Table 4.1(a).
- $V_k = \sum_{j \in I} \alpha_j$ be the value of the left part of equation (4.9.1) at iteration k of the dual problem (D(δ)).

In this heuristic we assume that β_t is a binary variable, i.e. $\beta_t \in \{0,1\}$ for all t. The heuristic starts with a fixed δ. It finds a feasible solution to problem (D(δ)) by assigning values to each α_j and β_t for all j and t, so that constraints sets (4.9.2) and (4.9.3) are satisfied.

Initial value of β_t. Remember that β_t is the dual variable corresponding to constraint t in set (4.8.2) when (P(δ)) is relaxed. $C_t \leq M$ means that there are at most M jobs that might be processed at time t, consequently constraint (4.8.2) is always satisfied in problem (D(δ)). Hence, its corresponding dual variable β_t is equal to zero in the optimal solution. For each t, the initial value of β_t is computed as follows:

$$\beta_t = 1 \text{ if } C_t > M \text{ and } 0 \text{ otherwise} \qquad (4.10)$$

The basic idea in the dual heuristic is to fix all β_t to all their potential highest values, that is 0 for those that the corresponding constraint in set (4.8.2) is always satisfied and 1 for the others. We then, iteratively, try to set them one by one to their lower possible values, to decrease W.

Setting a strictly positive variable β_t to zero. The heuristic proceeds in an iterative way to reduce W. Reduction of W may be achieved by setting a strictly positive variable β_t to zero since each β_t has an *"M"* weight in the calculation of W. Let $V_{k-1} = \sum_{j \in I} \alpha_j$ be the left part of W at step k-1 of the algorithm. Let $V_{k,t}$ be the left part of W value at step k after *tentatively* setting some strictly positive β_t to zero. If $V_{k,t} - V_{k-1} < M$ than the W will decrease by $[M - (V_k - V_{k-1})]$. We then choose t such that β_t gives the higher decrease of W. So,

$$t^* = \arg \max_{t:\beta_t>0} [M - (V_{k,t} - V_{k-1})]^+ \qquad (4.11)$$

The variable β_{t^*} to be set to zero is chosen so that the decrease of W is maximized. Note that, in order to decrease W, the increase of V_k should be smaller than M.

Computation of α_j. To satisfy constraints set (4.9.2) given values of β_t for all t, we compute α_j as follows:

$$\alpha_j = \max_{t \in \{r_j+1,\ldots,\ \delta - q_j - p_j + 1\}} [\, p_j - \sum_{s=t}^{t+pj-1} \beta_s]^+ \qquad \forall\, j \in I \qquad (4.12)$$

The dual step (DHB). Each time we come up with an improved W, we check whether $W < S^{p_j}$. If this is the case, $\delta+1$ is a lower bound on δ^* because we are sure that δ does not allow processing all jobs. We can then increase δ and restart the heuristic from the beginning with the new value of δ. If $W \geq S^{p_j}$, we decrease δ and restart the heuristic from the beginning with the new value of δ. The dual heuristic stops when $W < S^{p_j}$ or W can no longer be decreased, i.e. $M < (V_{k,t^*} - V_{k-1})$. Figure 4.7 gives the dual step of the heuristic for a fixed δ.

DHB
k = 0; with the given δ, compute β_t for all t, α_j for all j and V_0;
repeat
    k = k + 1;
    for each t with β_t=1 compute $V_{k,t}$ using (4.12) as if β_t was set to zero ;
    $t^* = \arg \max_{t:\beta_t>0} [M - (V_{k,t} - V_{k-1})]^+$;
    $\beta_{t^*} = 0$, $V_k = V_{k,t^*}$;
    compute α_j for all j according to (4.12) and W;
until $((V_k - V_{k-1}) \geq M)$ or $(W < S^{p_j})$;

Figure 4.7: Dual step (DHB)

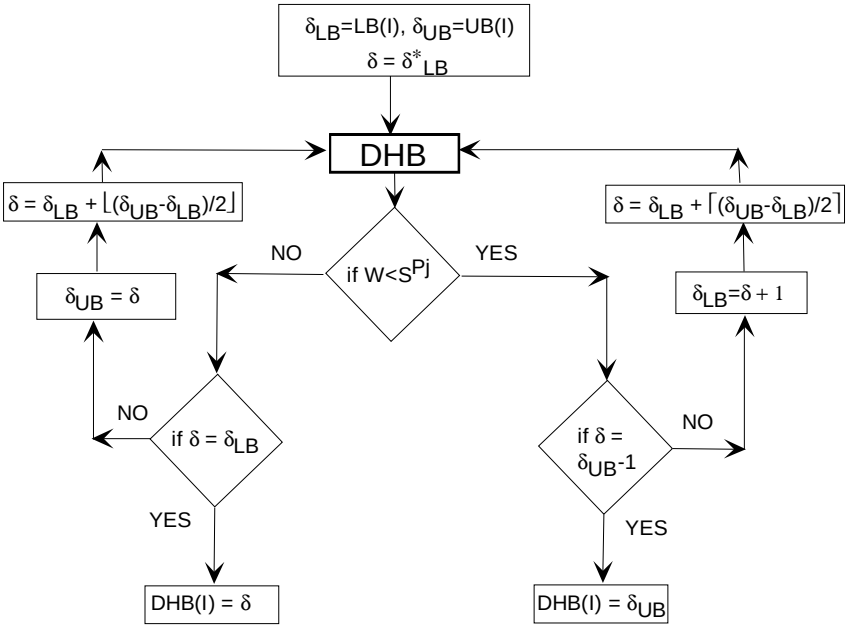


Figure 4.8: Diagram of the overall dual heuristic based lower bound

The diagram depicted in Figure 4.8 shows how the overall heuristic including the dual step and the bisection step. When the algorithm stops, DHB(I) is retained as a lower

bound. $[\delta_{LB},\delta_{UB}]$ is the current interval within which DHB seeks for the lower bound δ. At the beginning, δ_{LB} is equal to the best known lower bound LB(I) and δ_{UB} is equal to the best known upper bound UB(I). Note that we start by checking the lower limit of the initial bound interval $[\delta_{LB}, \delta_{UB}]$. This operation might save run time if the initial lower bound δ_{LB} is already the final one that would be found by DHB. This was the case of a large number of tested problem instances.

Example 4.3 shows how DHB(I) builds a lower bound, where M = 2 and the initial bound interval is [6, 8]. The dual corresponding constraints are given in table 4.1. In this table row and column δ=v appear only if the being checked value is larger than or equal to v.

Example 4.3:

j	r_j	p_j	Q_j	$6-q_j$	$7-q_j$	$8-q_j$
1	0	2	3	3	4	5
2	0	2	3	3	4	5
3	1	2	3	3	4	5
4	1	2	3	3	4	5

		t=1	t=2	δ=6 t=3	δ=7 t=4	δ=8 t=5	p_j ≥ 2
	α_1 +	β_1 +	β_2				≥ 2
δ=6	α_1 +		β_2 +	β_3			≥ 2
δ=7	α_1 +			β_3 +	β_4		≥ 2
δ=8	α_1 +				β_4 +	β_5	≥ 2
	α_2 +	β_1 +	β_2				≥ 2
δ=6	α_2 +		β_2 +	β_3			≥ 2
δ=7	α_2 +			β_3 +	β_4		≥ 2
δ=8	α_2 +				β_4 +	β_5	≥ 2
δ=6	α_3 +		β_2 +	β_3			≥ 2
δ=7	α_3 +			β_3 +	β_4		≥ 2
δ=8	α_3 +				β_4 +	β_5	≥ 2
δ=6	α_4 +		β_2 +	β_3			≥ 2
δ=7	α_4 +			β_3 +	β_4		≥ 2
δ=8	α_4 +				β_4 +	β_5	≥ 2
δ=6	C_t =	2	6	4			X
δ=7	C_t =	2	6	8	4		X
δ=8	C_t =	2	6	8	8	4	X

Table 4.1:(a) Dual constraints when δ=6, 7 and 8, row and column δ=v appear only if the checked value is ≥ v.

	k	Max. Decrease	t^*	α_1	α_1	α_1	α_1	β_1	β_2	β_3	β_4	β_5	β_7	W_k
δ'=6	0			1	1	0	0	0	1	1	X	X	X	6
δ'=8	0			1	1	0	0	0	1	1	1	1	X	10
	1	0	3	1	1	1	1	0	1	0	1	1	X	10
	2	2	5	1	1	1	1	0	1	0	1	0	X	8
	3	-2												
δ'=7	0			1	1	0	0	0	1	1	1	X	0	8
	1	0	3	1	1	1	1	0	1	0	1	X	0	8
	3	-2												

Table 4.1:(b) Bisection and dual steps for δ=6, 7 and 8

Table 4.1(b) gives the bisection and the dual steps so as to find the lower bound. DHB starts checking with δ=6. The algorithm stops before starting the first step because $W=6 < S^{p_j} = 8$. The algorithm can conclude that δ=6+1 is certainly a lower bound on the optimal

makespan. Bisection on the interval [7, 8] leads to the checking of δ=8. At step k=1 the maximum possible decrease of W is equal to 0 and is obtained by setting β_3 to zero. At step k=2 the maximum possible decrease of W is equal to 2 and is obtained by setting β_5 to zero. Since W_2=8 we can still check whether W can be decreased. At step k=3 the maximum possible decrease of W is negative and the algorithm stops. At this point we cannot say that δ=8 is a lower bound because we do not know if δ=7 is an infeasible solution. The algorithm now checks with δ=7. The algorithm stops at step 3 with W = 8 and concludes that δ=7 is a lower bound because δ=6 is an unfeasible solution and δ=7 is a feasible one, i.e. $W=8=S^{p_j}$ when δ=7.

4.2 IMPLEMENTATION

The critical operation in DHB is the computation of α_j. At each step k, in order to select t* that gives the largest decrease of W, we need to compute $V_{k,t}$ for all $\beta_t>0$, which requires the computation of all α_j, $j\in I$, according to (4.12). Let $T = \max_{j\in I}[\delta-q_j-p_j-r_j+1]$ be the maximum number of possible starting dates for any job and $Pmax = \max_{j\in I} p_j$ be the maximum processing time. According to equation (4.12), the theoretical worst time complexity of calculating one α_j is of O(TPmax) because we maximize over (maximum) T and we sum over (maximum) Pmax. A first improvement is obtained by maintaining an array "*A*" giving the cumulative values of β_t, t=1, …, T, i.e. $A_t = A_{t-1} + \beta_t$ with t=2, …, T and $A_1 = \beta_1$. The array *A* helps to reduce the worst time complexity to O(T), because we no longer need to sum over s=t,…,(t+p_j-1). For each j and t we obtain in constant time the value of $[p_j - \sum_{s=t}^{t+pj-1}\beta_s]$ which is equal to $[p_j - A_{t+pj-1} - A_t]$.

Let β_{t^*} be the variable we want tentatively set to zero. From step k-1 to step k, α_j may only increase (note p_j is a constant and any β_t might only decrease). We can maximize only over $t\in\{t^*-p_j+1, \ldots, t^*\}$ with $r_j+1 \le t \le \delta-q_j$ instead of over $t\in\{r_j+1, \ldots, \delta-q_j-p_j+1\}$ because t* only appears in these terms. Example (4.4) presents a job j with p_j=2, r_j=0 and $(\delta-q_j-p_j+1)$=5. We can see that only constraints s=2 and s=3 (rows 2 and 3) might change the value of α_j when we set β_3 to zero.

Example 4.4: job j constraints in set (4.9.2) in formulation (**D(δ)**)

Constraint s		t=1	t=2	t^*=3	t=4	t=5	
1	α_j +	β_1 +	β_2				$\ge p_j$
2	α_j +		β_2 +	β_3			$\ge p_j$
3	α_j +			β_3 +	β_4		$\ge p_j$
4	α_j +				β_4 +	β_5	$\ge p_j$

To compute $V_{k,t}$, equation (4.12) can then be replaced by the following equation (4.13) where α_{jk} is the value of α_j at the step k of the dual algorithm (k starting from 1), and α_{j0} is computed using equation (4.12).

$$\alpha_{jk} = \text{Max}\{\alpha_{jk-1}, \max_{s\in\{\max(r_j+1,t^*-p_j+1),\ldots,\min(\delta-q_j,t^*)\}}[p_j - (A_{s+p_j-1}^{k-1} - A_s^{k-1})]^+\} \;\forall\, j\in I \quad (4.13)$$

When considering the reduction of β_t, all α_j can then be recomputed in O(nPmax) and therefore the complexity of selecting the best t, (t*), is O(nTPmax). In order to stop the heuristic, and in the worst case, we need to set all the β_t to zero. Hence the theoretical worst time complexity of the dual based heuristic is of $O(nT^2Pmax)$. The practical time complexity strongly depends on the values of the release dates and tails of jobs, and is lower than the theoretical one. Numerical tests will show that DHB outperforms (gives the best lower bound most of the times) all known lower bounds, see Section 9.

The second new lower bound we introduce in this chapter is the Partially Preemptive Bound (PPB). It produces a better lower bound than the pure Preemptive Bound (PB). Before introducing PPB, we first recall how to build a preemptive schedule using Horn's algorithm (Horn (1974)).

5. PREEMPTIVE BOUND (PB)

The minimum makespan of the preemptive schedule is a lower bound (called PB) on the non-preemptive one. A preemptive schedule has the following properties. (i) A job cannot start before its release date. (ii) While being processed on some machine, a job can be stopped and assigned to any other machine or reassigned later on the same machine.

Horn (1974) proposes a flow algorithm that runs in $O(n^3)$ to test whether a certain trial value for PB is feasible or not. Like in the case of DHB, PB can be found using binary search. Labetoulle et al., (1984) show that the outcome of the preemptive problem can be determined by applying a bisection procedure that makes at most $\min\{n^2, \log(n*Pmax)\}$ calls at Horn's algorithm, which needs $O(n^3)$ per check. Vandevelde et al. (1995) show that there exists upper and lower bounds on PB that differ at most by $\left(M - 1/M\right)*Pmax$, which implies that one can apply Horn's algorithm at most $\lceil \log Pmax \rceil$. Before presenting Horn's corresponding network flow problem, we give a mathematical formulation for the preemptive schedule.

5.1 MATHEMATICAL FORMULATION

In order to formulate the preemptive problem we can relax the non-preemptive formulation (P(δ)) by relaxing the continuous processing of jobs imposed by constraints sets (4.8.1) and (4.8.2). Let $y_{jt} = 1$ if job j is being processed in period t and 0 otherwise

$$(\overline{P}(\delta))\quad \overline{P}(\delta) = \max \sum_{j\in I} \sum_{t=r_j+1}^{\delta-q_j} y_{jt} \quad \text{s.t.} \qquad (4.14.0)$$

$$\sum_{t=r_j+1}^{\delta-q_j} y_{jt} = p_j \qquad \forall\, j\in I \qquad (4.14.1)$$

$$\sum_{j\in I} y_{jt} \le M \qquad \forall\, t \qquad (4.14.2)$$

$$y_{jt} \in \{0,1\} \qquad \forall\, j\in I, \forall\, t \qquad (4.14.3)$$

$$(\overline{U})\quad \overline{\delta}^* = \text{Min } \delta \quad \text{s.t.} \qquad (4.14.4)$$

$$\overline{P}(\delta) = \sum_{j \in I} p_j \qquad (4.14.5)$$

Constraints set (4.14.1) assigns the total processing times of each job j. It is a relaxation of constraints set (4.8.1). The assignment can be done in a preemptive way since no constraint imposes continuous or non-preemptive processing of a job. Constraints set (4.14.2) imposes an upper bound on the number of the used machines at any time t. Constraints set (4.14.2) is a relaxation of constraints set (4.8.2). Here, since preemption is allowed we just need to look at each period t and limit the number of used machines to the number of the available ones, i.e. M.

Formulation ($\overline{P}(\delta)$) answers the following question: does a given value of δ allow the processing of all jobs in the preemptive case? In other words, is $\overline{P}(\delta) = \sum_{j \in I} p_j$? Knowing whether δ permits processing all jobs or not, the second problem is to find the smallest value of δ allowing the scheduling of all jobs in the preemptive case. Formulation ($\overline{U}$) finds the minimum δ such that $\overline{P}(\delta) = \sum_{j \in I} p_j$.

Formulation ($\overline{P}(\delta)$) is a transportation problem. It has an integral polyhedron, that is all problem solutions are integer, i.e. in this problem formulation $y_{jt} \in \{0,1\}$ is equivalent to $y_{jt} \geq 0$ and $y_{jt} \leq 1$. So relaxing the integrality constraints set (4.14.3) does not change the problem solution.

One can use a polynomial maximum flow algorithm to solve it. However, the flow problem corresponding to problem ($\overline{P}(\delta)$) has a large size because of the time indexed formulation. One can reduce the size of such flow problem by grouping the assignment of some jobs, i.e. (4.14.1), into times interval. The following subsection presents such improvement, which has been introduced by Horn (1974).

5.2 NETWORK FLOW PROBLEM OF THE PREEMPRIVE CASE

Before introducing the corresponding transportation problem, we first define a set T composed of the release dates r_j and the values (δ-q_j) of all jobs. Note that T is a set and contains no duplicated elements, i.e. each element appears only once.

$$T = \{u_{j1}, u_{j2} \mid u_{j1} = r_j \text{ and } u_{j2} = \delta - q_j, \forall j \in I\}$$

T is sorted in a non-decreasing order and |T|-1 time intervals are built as follows. For an illustration see later Example 4.5 and Figure 4.9.

$$T = \{t_1, t_2, \ldots, t_{|T|}\} \text{ so that } t_1 \leq t_2 \leq t_3 \leq t_4 \leq \ldots \leq t_{|T|}$$
$$V1 = [t_1, t_2],\ V2 = [t_2, t_3],\ V3 = [t_3, t_4],\ \ldots,\ V|T|-1 = [t_{|T|-1}, t_{|T|}]$$

With the available jobs and these time intervals, we build a bipartite graph or network made up of two layers of nodes, L1 and L2. Nodes of layer L1, or the left-hand side nodes, correspond to jobs $j \in I$, one node per job j. Nodes of L2, or the right-hand side nodes, correspond to the time intervals V_k, k=1, …, |T|-1, one node per time interval V_k. For each node (job) j, we add an incoming arc of flow p_j. The flow on this arc, which is the processing time of job j, represents the in-flow into the corresponding node. For each

node (job) $j \in I$, and each node V_k, k=1, …, |T|-1 we add one directed arc linking node j to node $V_k=[a, b]$ if $a \geq r_j$ and $b \leq \delta - q_j$. The flow on this arc is bounded from above by c(j, k) = min $[p_j, b-a]$ for each interval V_k. The flow on such arc represents the possible assignment of a portion p_{kj}, of the processing time of job j to the time interval V_k, with $p_{kj} \leq c(j, k)$ and $\sum_{k=1}^{|T|-1} p_{kj} = p_j$. The length of each time interval $|V_k| = (b - a)$ multiplied by the number of available machines M is an upper bound on the total amount of processing time that can be assigned and performed during this interval. Hence, we add an outgoing arc to each node representing each time interval V_k whose flow is upper bounded by $|V_k|*M$. Example 4.5 and Figure 4.9 show how the time intervals are built up. Figure 4.10 depicts the corresponding network flow problem. T = {1, 2, 4, 5, 6} and the corresponding time intervals are $V_1=[1, 2]$, $V_2=[2, 4]$, $V_3=[4, 5]$ and $V_4=[5, 6]$.

Example 4.5: $\delta = 8$ and M = 2

j	r_j	p_j	q_j	$\delta - q_j$
1	1	3	4	4
2	2	2	3	5
3	4	2	2	6

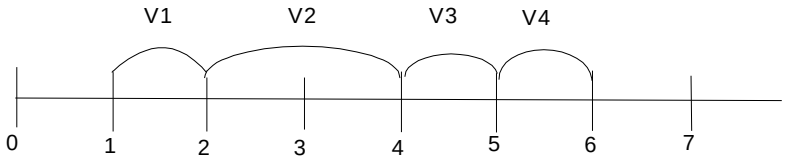


Figure 4.9: Time intervals defined by r_j and $\delta - q_j$, $j \in I$ of example 4.5

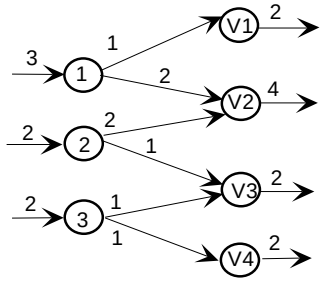


Figure 4.10: A network flow problem, derived from example 4.5

This flow problem uses a trial value δ and allows checking whether δ permits the processing of all jobs in the preemptive case. In other words, is the maximum flow $\bar{F}(\delta) = S^{p_j} = \sum_{j \in I} p_j$?

If $\bar{F}(\delta) < S^{p_j}$, we can conclude that δ+1 is a lower bound on the optimal makespan δ*, because δ does not allow producing a preemptive schedule. As in the case of the dual based heuristic DBH, see Section 4, using bisection on some lower and upper bounds interval $[\delta^*_{LB}, \delta^*_{UB}]$ on the optimal makespan δ* allows finding the preemptive bound. Let PB be the algorithm that, using some trial value δ, allows computing the corresponding maximum flow $\bar{F}(\delta)$. In order to find the preemptive bound one can then use the diagram in Figure (4.8) to do bisection on $[\delta^*_{LB}, \delta^*_{UB}]$ by replacing DHB and W with PB and $\bar{F}(\delta)$, respectively.

A tighter lower bound than PB may be obtained by constraining the preemptive assignment of jobs, i.e. solving a tighter relaxation than the preemptive relaxation. In the next section we show how to build a partially preemptive bound.

6. PARTIALLY PREEMPTIVE SCHEDULE (PPB)

6.1 MATHEMATICAL FORMULATION

This formulation is a relaxation of the problem of building a non-preemptive schedule, i.e. (P(δ)). It resembles formulation ($\overline{P}(\delta)$) but constraint set (4.14.1) has been modified to constrain the preemption of jobs. Let $y_{j,t} = 1$ if job j is being processed in period t and 0 otherwise. This time indexed formulation has been introduced by Sousa and Wolsey (1992) for the single machine problem.

$$(\overline{\overline{P}}(\delta))\quad \overline{\overline{P}}(\delta) = \max \sum_{j\in I} \sum_{t=r_j+1}^{\delta-q_j} y_{j,t} \qquad (4.15.0)$$

$$\text{s.t.} \quad y_{j,t+r_j} + y_{j,t+r_j+p_j} + y_{j,t+r_j+2p_j} + y_{j,t+r_j+3p_j} + \ldots + y_{j,t+r_j+kp_j} = 1 \qquad (4.15.1)$$

$$\forall\, j,\ \forall\, t = 1, \ldots, p_j \text{ and}$$

$$k \text{ is the largest integer such that } t+r_j+kp_j \le \delta - d_j$$

$$\sum_{j\in I} y_{j,t} \le M \qquad \forall\, t \qquad (4.15.2)$$

$$y_{jt} \in \{0,1\} \qquad \forall\, j,\ \forall\, t \qquad (4.15.3)$$

$$(\overline{\overline{U}}) \quad \overline{\overline{\delta^*}} = \text{Min } \delta \quad \text{s.t.} \qquad (4.15.4)$$

$$\overline{\overline{P}}(\delta) \ge \sum_{j\in I} p_j \qquad (4.15.5)$$

Constraint set (4.15.1) states that each job must have been assigned a total of p_j periods or time units, each constraint (j, t) assigns one time-unit of the processing time p_j of job j. Constraint set (4.15.2) imposes an upper bound on the number of used machines at any time t. Constraint set (4.15.1) also presents how the assignment of the time units should take place. Actually, in the non-preemptive case, if a job starts processing at time $t+r_j$, its completion time occurs in period $(t+r_j+p_j-1)$. Because of formulation (4.15.1), and by keeping in mind what happens in the non-preemptive case, a job to which one time-unit of its processing time has been assigned at time $t+r_j$, i.e. $y_{j,t+r_j} = 1$, will be completed by time $(t+r_j+p_j)$. Therefore, $y_{j,t+r_j+p_j}$ can be set to zero. Furthermore, this job could not be processed at time $(t+r_j-p_j)$. The same reasoning can be made for $(t+r_j+p_j)$, $(t+r_j+2*p_j)$, etc.

This constraint can, intuitively, be understood as follows. Suppose the processing time p_j of job j is divided into p_j time-units p_{jk}, k=1, …, p_j. Ideally (in the non-preemptive case), when p_{j1} is assigned in period t each of the remaining time units p_{jk}, k=2,…, p_j of job j is assigned in period t+k-1. Here, we simply impose a gap or a time limit with length p_j between the possible assignments of one time unit p_{jk}. That is a time unit p_{jk} could be

assigned only in one of the following periods $k+r_j$, $k+r_j+p_j$, $k+r_j+2*p_j$, $k+r_j+3*p_j$, etc, with $k=1,\ldots, p_j$. Note that this constraint is always respected in the non-preemptive case.

The above formulation is more constrained than the purely preemptive case. Consider that one time unit of job j is being processed at time t, in the purely preemptive case, the next time-unit could then be assigned at any time (t + i), i = 1, 2, 3, …, etc. However, in the partially preemptive case the assignment is more constrained and the remaining processing time units of j will never be assigned at time (t + i), i = p_j, $2*p_j$, $3*p_j$, etc. In other words, the preemptive schedule is constrained by adding a constraint, which is always satisfied in the non-preemptive case. This intermediate formulation proposes then a schedule with a length which is between the length of purely preemptive schedule and the length of the non-preemptive schedule.

Example 4.6 illustrates how the assignment of the processing time of a job might occur within a time interval [1, 5] using formulation (4.15.1) and where $p_j = 2$. The first constraint imposes that one unit of processing time must be assigned to one of the periods 1, 3, 5. The second constraint assigns the second processing time unit to either period 2 or period 4. So, for example if $y_{j1} = 1$, the assignment of the second time unit of p_j to period 3 and 5 is allowed in the preemptive case, but not in the partially preemptive one.

Example 4.6: Possible assignment of job j in [1, 5], $p_j = 2$ using (4.15.1)

Constraint	t=1	t=2	t=3	T=4	t=5
1	X		X		X
2		X		X	

Like ($\overline{P}(\delta)$), formulation ($\overline{\overline{P}}(\delta)$) allows to check whether $\overline{\overline{P}}(\delta) = \sum_{j\in I} p_j$. Knowing whether δ permits processing all jobs in the partially preemptive case, the second problem is to find the smallest value of δ allowing the scheduling of all jobs in the partially preemptive case. Formulation ($\overline{\overline{U}}$) finds the minimum δ such that $\overline{\overline{P}}(\delta) = \sum_{j\in I} p_j$.

Formulation ($\overline{\overline{P}}(\delta)$) is also a transportation problem. It has an integral polyhedron. Relaxing the integrality constraints set (4.15.3) does not change the problem solution. One can use a polynomial maximum flow algorithm to solve it. The derived network flow problem permits us to check, for a given number of available machines, whether a given limit of time δ allows building a partially preemptive schedule. Such derived network flow problem is presented next.

6.2 NETWORK FLOW PROBLEM

Let T be the set of available time-periods within which we need to check the feasibility of building a partially preemptive schedule. Let $v = 1+ \min_{j\in I} r_j$.

$$T = \{ t \mid t = v, v+1, v+2, \ldots., \delta- \min_{j\in I} q_j \}$$

With the elements of the set T and with the processing times of jobs, we build a bipartite graph formed with two layers of nodes, L1 and L2. Nodes of layer L1, or the left-hand

side nodes, correspond to nodes (j, i) with $j \in I$ and i = 1, …, p_j where node (j, i) corresponds to the i^{th} time processing of job j. Nodes of L2, or the right-hand side nodes, correspond to the possible time periods available for processing the available jobs, one node (t) for each time-period, $t \in T$.

For each node (j, i), we add one incoming directed arc with a flow of one unit corresponding to one unit of the processing time of job j. We link each node (j, i) of L1 to all the L2 nodes in the set $\{(i+r_j), (i+r_j+p_j), (i+r_j+2*p_j), \ldots,\}$. The flow on each arc is bounded from above by 1. The inflow of 1 unit into node (j, i), and the distribution of this unit to the nodes in $\{(i+r_j), (i+r_j+p_j), (i+r_j+2*p_j),\ldots\}$ models exactly constraints set (4.15.1).

Additionally, to each L2 node $t \in T$, we put an outgoing arc whose flow is bounded by the available capacity in period t which is equal to the number of machines M. Figure 4.11 gives the network flow that can be derived from Example (4.5) using constraints set (4.15.1).

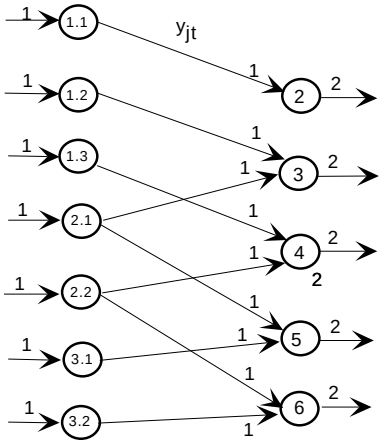


Figure 4.11: A network flow problem for building a partially preemptive schedule derived from Example 4.5

As in the case of PB, this flow problem uses a trial value δ and allows checking whether δ permits the processing of all jobs in the partially preemptive case. In other words, is the computed maximum flow $\overline{\overline{F}}(\delta) = S^{p_j} = \sum_{j \in I} p_j$?

If $\overline{\overline{F}}(\delta) < S^{p_j}$, δ+1 is a lower bound on the optimal makespan δ*. As in the case of the DBH and PB, using bisection on some lower and upper bounds $[\delta^*_{LB}, \delta^*_{UB}]$ on δ* allows finding the partially preemptive bound. Let PPB be the algorithm that using some trial value δ allows computing the maximum flow $\overline{\overline{F}}(\delta)$. One can then use the diagram in Figure 4.8 to do bisection on $[\delta^*_{LB}, \delta^*_{UB}]$ by replacing DHB and W with PPB and $\overline{\overline{F}}(\delta)$, respectively.

PB and PPB both rely on a maximum flow algorithm. We hereafter recall the "*Preflow-push"* algorithm we decided to use for solving the maximum flow problem, and present some implementation improvements allowing the running time reduction of "*Preflow-push"* when used in PB and PPB.

7. SOLVING THE MAXIMUM FLOW PROBLEM

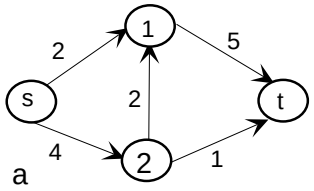
In this section we first recall and state the flow problem and some of its characteristics, like node distance labeling and admissible arcs concepts. We then recall the "*Preflow-push"* algorithm and its adaptation for bipartite graphs. We eventually present some improvements that allow reducing the practical running time of the "*Preflow-push"* when used for bipartite graphs. For a detailed study of maximum flow problems the reader can refer, for example, to Ahuja et al., (1993). The reader who is familiar with network problems and Preflow-push algorithms can skip subsections 7.1, 7.2 and 7.3.

7.1 MAXIMUM FLOW PROBLEM

Network representation. Let *G = (N, A)* be a capacitated network over node set N and directed arc set A, with a nonnegative integer capacity u_{ij} for any arc (i, j)∈ A and n= |N|. Two special nodes, s and t are, respectively, the source node from which the flow enters the network and the sink node, i.e. the only node, from which the flow exits the network. The maximum flow problem consists in finding the maximum flow that can be sent from node s to node t. Figure 4.12(a) gives an example of a capacitated network. Arcs with zero capacity are not depicted. Hence, for example u_{t1}=0 and u_{1t}=5.

If a flow x_{ij} transits by arc (i, j), or equivalently from node i to node j, the residual capacity r_{ij} of arc (i, j) is reduced by x_{ij}. In other words, the capacity of arc (i, j) is reduced and the capacity of arc (j, i) is increased. The residual capacity r_{ij}, of any arc (i,j)∈ A, represents the maximum additional flow that can be sent from node i to node j using the arcs (i, j) and (j, i). Two main characteristics can be noted: the unused capacity of arc (i,j) is equal to $u_{ij} - x_{ij}$, thus the current flow x_{ji} on arc (j, i) can be cancelled to increase flow from node i to node j. Consequently, $r_{ij} = u_{ij} - x_{ij} + x_{ji}$.

For any given flow x from s to t, the residual network G(x) is the network, represented by arcs having a positive residual capacity, i.e. $r_{ij} > 0$ ∀(i, j)∈ A. Figure 4.12(b) gives an example of a residual network after sending a flow of 2 units through nodes s-1-t. Arcs with zero capacity are not depicted. For example, since x_{1t}=2, x_{t1}=0 then r_{1t}=5–2+0=3 and r_{t1}=0-0+2=2. Note that r_{t1}=2 to represent the fact that the flow x_{t1} can be increased by 2, or equivalently x_{1t} reduced by 2.



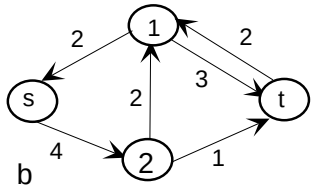


Figure 4.12: (a) Example of network with arc capacities
Figure 4.12: (b) Residual network with residual arc capacities

Distance Labels. In the residual network, each node i has an integral distance label d(i) representing a lower bound on its distance to node t. The exact distance label d(i) of any node i in the residual network is computed as the shortest distance from i to t, and satisfies the following:

$d(t) = 0;$
$d(i) \leq d(j) + 1 \quad \forall (i, j) \in A$, with $r_{ij} > 0$ in the residual network $G(x)$

In Figure 4.12 (b), the exact distance labels are: d(1)=1, d(2)=1, d(s)= d(2)+1=2.

Admissible arcs and admissible paths. In the residual network, an arc (i, j) is admissible if it satisfies the condition d(i) = d(j) + 1. Any path from node s to node t consisting entirely of admissible arcs is an admissible path. Note that by sending flow through admissible s-t paths, we are using the shortest path for transporting the flow from the source s to the sink node t.

Finding the maximum flow of a capacitated network is a polynomial problem of complexity $O(n^3)$. Many algorithms have been developed to solve it. *Preflow-push* algorithms are the most powerful techniques for solving the maximum flow problem both theoretically and empirically. Ahuja et al., (1997) have conducted an extensive study on a variety of maximum flow algorithms on several classes of networks, and concluded that the "*preflow-push"* algorithm outperforms all other tested algorithms. See also Ahuja et al., (1993).

7.2 PREFLOW PUSH ALGORITHMS

Unlike other methods, Preflow-push algorithms send flow (from the source to the sink) along single admissible arcs, while other algorithms, like the shortest augmenting path, send flow along admissible paths. We hereafter present the main idea behind the Preflow-push algorithms. See Ahuja et al., (1993) or (1997) for a detailed description and proof of correctness of these algorithms.

The operation of sending flow from node i to one of its adjacent nodes is called a *push*. The amount of flow e(i) stocked at node i is called the *excess* of node i. The basic operation of the algorithm is to select an active node (e(i) > 0) and try to remove its excess by pushing the flow to one of its neighbors. The flow is sent from node i only on admissible arcs, i.e. to node j with d(i)=d(j)+1. Hence the nodes to which the flow should be sent are the ones that are closer to the sink. Let A(s) be the arc set coming from source node s, i.e. $A(s)=\{(s, j)|\forall j \in N\}$. We hereafter give the *Preflow-Push* algorithm, given in Ahuja et al., (1993), which uses two routines: *preprocess* and *push/relabel*.

The preprocess routine initializes the algorithm by computing the exact initial distance labels and gives each node adjacent to node s a maximum positive excess, so that the algorithm can begin by selecting some node with a positive excess. Doing so, the *preprocess routine* saturates all arcs incident to node s, i.e. $(s, j) \in A(s)$. Setting d(s) = n, implies that the residual network contains no direct path from node s to node t, and we will never need to push flow from node s again. Indeed, d(s) is a lower bound on the length of the shortest path from s to t in G(x) and no directed path can contain more than n-1 arcs.

At this step the actual amount of the flow in the network is equal the sum of the capacity of those arcs coming from the source node, i.e. equal to what is called the *source cut* $S= \sum_{(s,j)\in A} u_{sj}$.

```
routine preprocess;
begin
    x = 0;
    compute the exact distance labels d(i);
    x_sj = u_sj for each arc (s, j) ∈ A(s);
    d(s) = n;
end;
```

```
routine push/relabel(i)
begin
    if the network contains an admissible arc (i, j) then
        push w = min{e(i), r_ij } units of flow from node i to node j
    else replace d(i) by min{d(j) + 1: (i, j) ∈ A(i) and r_ij >0};
end;
```

```
Algorithm Preflow-Push;
preprocess;
begin
    while the network contains an active node do
        select an active node i;
        push/relabel(i);
    end-while;
end;
```

The push/relabel routine(i) selects node i with e(i)>0 (selects an active node), and tries to push the flow w=min{e(i),r_{ij}} through an admissible arc (i, j)∈A, i.e. d(i)=d(j)+1 and r_{ij}>0. If no admissible arc (i, j) exists, i.e. d(j) ≥ d(i) and/or r_{ij} = 0 for all (i, j)∈A, which also means that the distance label d(i) of node i is no more valid, d(i) needs then to be updated and its distance to the sink node will increase. Therefore, the distance label of any node i is updated as follows: d(i) = min{d(j) + 1: (i, j) ∈ A(i)} only when node i is selected as an active node and no admissible arc exists to push its excess to some adjacent node. Hence, distance labels are always valid. Increasing the distance label of a node, with no admissible arc, creates at least one admissible arc. In Figure 4.13, the excess flow at node i cannot be pushed neither to node v nor to node k. The only way to push that flow is to update (increase) the distance label of node i as follows: d(i)=min{d(j)+1: (i,j)∈A(i)}=2.

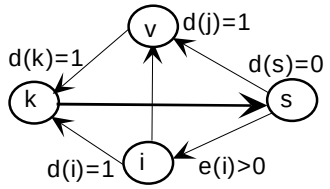


Figure 4.13: Example of a network with no admissible arc

The algorithm terminates when the network contains no active node, except at node s and node t, in which case a maximum flow x^*, equal to the flow that gets out from the network through node t, has been found. The flow x^* could be equal to the initial amount

of flow sent via the source node (equal to source cut $S = \sum_{j \in N} u_{sj}$). If $x^* < S$ the flow $D=S-x^*$ has been sent back to the source during the pushing process of the algorithm. Note also that the initial flow S is an optimal flow only if it leaves the network via node t. A necessary condition is that the sink cut $T = \sum_{(i,t) \in A} u_{it}$ is at least equal to the source cut S, i.e. large enough to let the S flow gets out of the network.

The Preflow-push algorithm has a worst time complexity of $O(n^2m)$ where n is the number of nodes and m is the number of arcs of the network, Ahuja et al., (1993).

Figure 4.14 gives an example illustrating the different steps of the Preflow-push algorithm so as to find a maximum flow. Step 1 shows the network after executing the preprocessing routine, i.e. pushing maximum flow to node 1 and 2 and computing the exact distance labels.

The algorithm stops at the end of step 4 when the network contains no active node, i.e. e(1) = e(2) = 0. At step 4, e(t)=6 means that the maximum flow found is equal to 6, and e(0)=1 means that from the initial pushed flow by the preprocessing routine, i.e. 7, one unit could not be pushed out of the network via t and has been sent back to s. Note that in step 3, in order to send e(1)= 1 from node 1 to node s the distance label of node 1 has increased to 5.

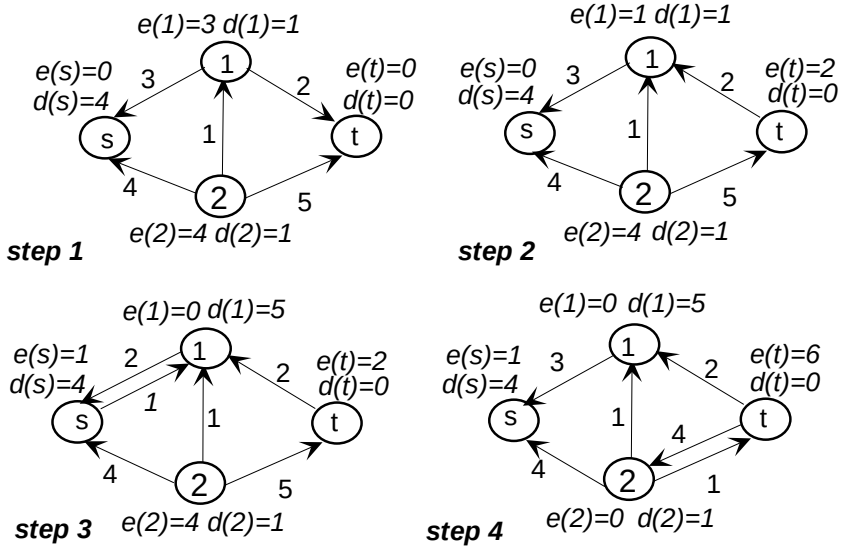


Figure 4.14: An example illustrating the Preflow-push algorithm

7.3 SPECIALIZED PREFLOW PUSH ALGORITHM FOR BIPARTITE NETWORKS

Bipartite networks have a nice structure. A specialization of the preflow-push algorithm has been developed for bipartite networks and yields algorithms with improved worst-case complexity. See Figures 4.10 and 4.11 for examples of bipartite graphs. Two major properties of the bipartite network can be put into use for reducing the worst-case complexity of the preflow-push algorithm.

Recall that the Preflow-push algorithm initializes the distance label of the source node to n where n is the number of nodes. While pushing flows through the network if some excess is blocked at node i and needs to be brought back to the source node, its distance label must be strictly greater than n, because we are pushing flows only on admissible arcs.

Let L1 and L2 be the sets of the left-layer nodes and right-layer nodes, respectively, in a bipartite network. In such representation node s and node t belong to L2 and L1, respectively. If |L1|<<<|L2| and while initializing the distance label of the source node s, we can set d(s) to 2*|L1| since the longest path from the source node to the sink node does not exceed 2*|L1| in a bipartite graph network.

The second improvement replaces the push/relabel(i) routine by a bipartite push/relabel(i) routine with $i \in L1$, where the single arc-push is replaced by a two-arc-push operation. This routine permits only the nodes in L1 to be active and by this way reduces the worst time complexity of the algorithm to $O(|L1|^2 m)$. The Preflow-push algorithm has been introduced by Goldberg and Tarjan (1986). The reader can refer to Ahuja et al., 1993 for more details on the two above-mentioned algorithms. The *push/relabel* routine is replaced as follows in the bipartite case algorithm.

```
Routine bipartite  push/relabel(i)   [i∈ L1]
begin
    if the network contains an admissible arc (i, j) then
        if the network contains an admissible arc (j, k) then
            push w = min{e(i), r_ij , r_jk } units of flow from node i to node k, through node j
        else replace d(j) by min{d(k) + 1: (j, k) ∈ A(j) and r_jk > 0 };
    else replace d(i) by min{d(j) + 1: (i, j) ∈ A(i) and r_ij >0};
end;
```

In the next section we introduce an improved version of the Preflow-push algorithm for bipartite graphs.

7.4 A NEW PREPROCESSING ROUTINE FOR BIPARTITE GRAPHS

The main characteristic of bipartite graphs is that they are made up of two layers of nodes. The main characteristic of the Preflow-push algorithm for bipartite graph is to maintain active only the nodes of the layer with smallest cardinality, that is to push the flow from active nodes through two arcs instead of one. The main characteristic of our new version of the Preflow-push algorithm for bipartite graph is to push the initial flow directly to the sink node instead of to the first layer of nodes only. In the Preflow-push, the preprocess routine initializes the algorithm by computing the exact initial distance labels and by pushing all the flows coming from the source node s to its neighbors, i.e. by saturating all arcs $(s,j) \in A(s)$. Our new version modifies the *preprocess routine* only.

The idea here is when building and initializing the network, each time we link three nodes i, j and t with directed arcs (i, j) and (j, t) we push the flow $w = \min\{e(i), u_{ij}, u_{jt}\}$ stocked in node i into node t. Note that this operation never relabels the nodes, and if the arc (i, j) and/or (j, t) are saturated it leaves the excess at node i intact. Once the graph is

built and initialized, we compute the exact distance labeling of all nodes and start the algorithm as in its classical version. We hereafter give the modified *preprocess routine*. Figure 4.15 gives an example illustrating the modified preprocess routine of the Preflow-push algorithm for bipartite graph. Figure 4.15(a) gives the network being built. Figure 4.15(f) gives the network at the end of the modified preprocess routine. At this step the algorithm proceeds as in the initial version. Note that with our version, at the beginning of the Preflow-push algorithm a flow of 5 units has gotten out from t already, and the network has a total excess of 2 units, i.e. 2=e(1)+e(2).

```
routine Modified-preprocess;
begin
    x = 0;
    For each arc (s, j)∈ A(s), push u_sj to node j and set e(i) = u_sj;
    For each pair of arcs (i, j) and (j, t)
          If e(i) ≠ 0
                push w = min{e(i), r_ij , r_jt } units of flow from node i to node t, through node j
     end-for;
    d(s) = n; d(t) =0;
    compute the exact distance labels d(i) for all i≠n and i≠0;
end;
```

Numerical tests (see Section 9) have shown that once this modified preprocess routine is completed a large amount of flow has already gotten out from the network before starting the push/relabel steps of the algorithm. Though the theoretical complexity has not changed, numerical tests show that this version of the Preflow-push algorithm for bipartite graphs consumes less computing time than the original one.

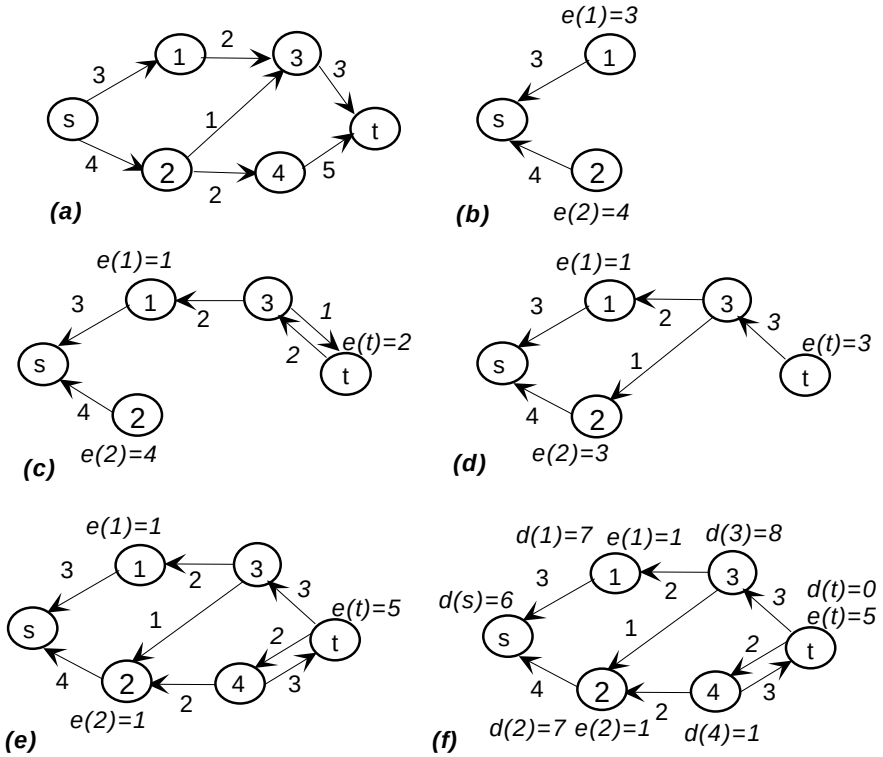


Figure 4.15: An example illustrating the modified preprocess routine of the Preflow-push algorithm for bipartite graphs

7.5 NECESSARY STOPPING CONDITIONS

In our network flow problems, used for finding the lower bound PB and PPB, we are only interested in checking the feasibility of flow equal to the source cut, and not in finding the maximum flow value when this source cut flow is infeasible. The following conditions reduce the running time complexity of the *Preflow push algorithm* when used in PB and PPB.

Condition 1. If the source cut is larger than the sink cut then the flow corresponding to the source cut is not a feasible flow.

Condition 2. While processing the Preflow-push algorithm, if the distance label of an active node is equal to or greater than n (i.e. the number of nodes) the flow corresponding to the source cut is not a feasible flow, i.e. if e(i)>0 and d(i)≥n then in the residual network there is no direct path from node i to node t. However, there is a direct path from node i to the source node, to which the flow e(i) will be sent back.

In Figure 4.15(f), d(1) = 7 > 6 means that the only way out from the network for excess e(1) is through node s. Indeed, an excess sent back to node s could never be reintroduced into the network, since all adjacent arcs to node s have been saturated at the beginning of the preprocess routine.

8. HEURISTIC FOR THE SUBSET BOUND

The idea behind the subset bound SSB(I) (see equation (4.5)) is to compute the set bound over all subsets of I and to retain the maximum value. The subset $V^* \subseteq I$ with $|V^*|>M$, (where M is the number of machines) leading to this value, i.e. SSB(I) = SB(V^*), is the optimal subset. Vandevelde et al., (1995) have shown that SSB can be calculated in $O(n^2)$ without enumerating all subsets of I. Here, we present a Heuristic for SSB (HSSB) which is computed in O(nM) where n is the number of jobs. The idea behind HSSB is to build a faster algorithm which guarantees a very good approximation of SSB. Actually HSSB does not guarantee finding the optimal subset bound SSB(I), i.e. HSSB(I)≤SSB(I) (See later on Example 4.8 in Subsection 8.5). However, in our numerical tests, the final version of HSSB always gives the same value as SSB, (see Section 9) because it always finds the optimal subset V, i.e. SB(V)=SSB(I).

The main idea of HSSB is to proceed iteratively as follows: it first starts by building SB(I) where all jobs in set I are used. At each step k a job j is removed from set $V_k \subseteq I$ where V_0=I. Eventually it finds a subset V_{k^*} such that $|V_{k^*}|=|I|-k$, SB(I) ≤ SB(V_{k^*}) ≤ SSB(I) and where SB(V_{k^*}) is as large as possible. The overall heuristic *HSSB* is given hereafter.

HSSB. We first build the set bound SB(I), which we then try to improve in an iterative way. At each iteration k=1, …, n-M-1 we discard one job $j_k^* \in V_{k-1}$ (V_0 = I) such that the set bound which can be built with the leftover jobs is maximized, i.e.

$SB(V_k)= \max_{j\in V_{k-1}} SB(V_{k-1}\backslash\{j\})$. At each iteration k the heuristic obtains a new set bound $SB(V_k)$ using subset V_k after removing job j_k^* from set V_{k-1}, i.e. $V_k=V_{k-1}\backslash\{ j_k^* \}$, $V_k\subseteq I$ and $|V_k|=n - k$, where n is the number of jobs. The heuristic stops when the total number of the leftover jobs in V_k is equal to M + 1. Indeed, if $|V_k| \leq M$ the lower bound over V_k is equal to the optimal makespan and to the job bound $JB(V_k)$ because each machine processes one job at most. Since at each iteration k we obtain a new lower bound $SB(V_k)$ built using a subset of jobs, the HSSB retains the maximum as a final lower bound.

$$HSSB(I) = SB(V_{k^*}) = \max_{k=1,\ldots,n-m-1} SB(V_k) \quad (4.16)$$

The proof that HSSB(I) delivers a lower bound on our original problem is trivial. HSSB is just a set bound on the subset $V_{k^*}\subseteq I$, (see Section 2). The main critical operations, which determine the computational complexity of this heuristic, are the selection of the job $j_k^* \in V_{k-1}$ and the computation of $SB(V_k)$, which we improve in the following.

8.1 SELECTION OF j_k^*

At each iteration k we maintain three sets of jobs $R(V_k)$, $Q(V_k)$ and $RQ(V_k)$. $R(V_k)$ is the set of jobs where only the release dates are used in the computation of $SB(V_k)$. $Q(V_k)$ is the set of jobs where only the tails are used in the computation of $SB(V_k)$ and $RQ(V_k)$ is the set of jobs where both release dates and tails are used in the computation of $SB(V_k)$. $SB(V_k)$ is computed as follows:

$$\begin{aligned} SB(V_k) &= \lceil (r_{[1]} +\ldots+ r_{[M]} + q_{[1]} +\ldots+ q_{[M]} + \sum_{j\in V_k} p_j) / M \rceil \\ &= \lceil \sum_{j\in R(V_k)} r_j + \sum_{j\in Q(V_k)} q_j + \sum_{j\in RQ(V_k)} (r_j + q_j) + \sum_{j\in V_k} p_j) / M \rceil \\ &\text{with } |R(V_k)| + |RQ(V_k)| = M = |Q(V_k)| + |RQ(V_k)| \end{aligned}$$

At iteration k+1, the job to be discarded is selected only from those in set $S_k=R(V_k)\cup Q(V_k)\cup RQ(V_k)$. Indeed, at iteration k, only jobs in set S_k have the potential of increasing the lower bound $SB(V_k)$, i.e. only the M smallest release dates and the M smallest tails of the jobs in V_k are used in the computation of $SB(V_k)$.
Selecting a job belonging to S_k instead of V_k, reduces drastically the computation complexity of HSSB, since $|S_k| \leq 2*M$. The job j_k^* to be discarded from V_{k-1} at iteration k is selected as follows in a greedy way:

$$j_k^* = \arg\max_{j\in S_{k-1}} SB(V_{k-1}\backslash\{j\}) \quad (4.17)$$

8.2 COMPUTATION OF $SB(V_K)$

Actually, we do not need to compute SB at each iteration k but we just need to update $SB(V_{k-1})$ to come up with $SB(V_k)$. Since at iteration k we maintain the three sets of jobs

$R(V_k)$, $Q(V_k)$ and $RQ(V_k)$ and knowing the job to be discarded j_k^*, the updating of $SB(V_{k-1})$ is easily carried out by updating the numerator of $SB(V_{k-1})$. Let T_k be the numerator of $SB(V_k)$ at iteration k: $T_k = r_{[1]} + r_{[2]} + \ldots + r_{[M]} + q_{[1]} + q_{[2]} + \ldots + q_{[M]} + \sum_{i \in V_k} p_i$

In order to remove a job j_k^* from the set V_{k-1} and update T_{k-1} to come up with T_k we proceed as follows:

- If $j_k^* \in R(V_{k-1})$ then $T_k = T_{k-1} + r_i - r_{j^*} - p_{j^*}$
- If $j_k^* \in Q(V_{k-1})$ then $T_k = T_{k-1} + q_j - p_{j^*} - q_{j^*}$
- If $j_k^* \in RQ(V_{k-1})$ then $T_k = T_{k-1} + r_i + q_j - r_{j^*} - p_{j^*} - q_{j^*}$

where i and j are the jobs having respectively the $M+1^{th}$ smallest release date and the $M+1^{th}$ smallest tail in V_{k-1}.

Let $T_{k-1}(j)$ be the value of the numerator of $SB(V_{k-1}\backslash\{j\})$ and $Incr(V_{k-1}\backslash\{j\}) = T_{k-1}(j) - T_{k-1}$ be the potential increase of T_{k-1} when we remove j from set V_{k-1}. The following equation redefines the job j_k^* to eliminate from V_{k-1} as:

$$j_k^* = \arg\max_{j \in S_{k-1}} T_{k-1}(j) = \arg\max_{j \in S_{k-1}} Incr(V_{k-1}\backslash\{j\}) \qquad (4.18)$$

The updating of T_k requires finding the $M+1^{th}$ smallest release date and the $M+1^{th}$ smallest tail of V_{k-1}. It also implies the updating of sets $R(V_k)$, $Q(V_k)$ and $RQ(V_k)$ so they can be valid at the next iteration k + 1 for the computation of T_{k+1}. These two operations can be handled, respectively, in a constant time and in O(M).

Finding the $M+1^{th}$ smallest release date and the $M+1^{th}$ smallest tail of V_k

In order to find, at each iteration k, and in constant time, the $M+1^{th}$ smallest release date and the $M+1^{th}$ smallest tail of the jobs in set V_{k-1}, we maintain two sets Ir_k and Iq_k such that $Ir_0=I$ and $Iq_0=I$. At the beginning of the heuristic, these two sets are sorted in a non-decreasing order of the release dates and the tails, respectively, which can be done in O(*nlogn*). Each time a new release date r_i is included in T_k (see above the computation of T_k) the corresponding job i is removed from set Ir_k (i.e. $Ir_k = Ir_{k-1}\backslash\{i\}$). In the same way, each time a new tail q_j is included in T_k the corresponding job j is removed from set Iq_k (i.e. $Iq_k=Iq_{k-1}\backslash\{j\}$). Removing an element from sets I_{rk} and I_{qk} is done in constant time because we always first include in T_k the smallest value ($M+1^{th}$ smallest release date) in Ir_k and/or the smallest value ($M+1^{th}$ smallest tail) in Iq_k. See later Example 4.7.

Updating of $R(V_k)$, $Q(V_k)$ and $RQ(V_k)$

Let i and j be the jobs whose release date and tail, respectively, need to be included in the computation of T_k. Note that the job i whose release date only is to be included in T_k, might already have its tail included in T_{k-1}, $i \in Q(V_{k-1})$. In the same way, the job j whose tail only is to be included in T_k, might already have its release date included in T_{k-1}, $j \in R(V_{k-1})$. The third case is where both release date and tail of a job are to be included in T_k, i.e. i and j represent the same job and $i \notin S_{k-1}$. $R(V_k)$, $Q(V_k)$ and $RQ(V_k)$ are updated as follows:

If i = j then $RQ(V_k) = RQ(V_{k-1}) \cup \{i\}$
else

If $i \in Q(V_{k-1})$ then	$Q(V_k) = Q(V_{k-1}) \setminus \{i\}$ and $RQ(V_k) = RQ(V_{k-1}) \cup \{i\}$
else	$R(V_k) = R(V_{k-1}) \cup \{i\}$
If $j \in R(V_{k-1})$ then	$R(V_k) = R(V_{k-1}) \setminus \{j\}$ and $RQ(V_k) = RQ(V_{k-1}) \cup \{j\}$
else	$Q(V_k) = Q(V_{k-1}) \cup \{j\}$

Figure 4.16 gives the skeleton of the HSSB heuristic. The section of j_k^* and the updating of T_k, $R(V_k)$, $Q(V_k)$, $RQ(V_k)$, Ir_k as well as Iq_k have been described above.

```
HSSB bound
V0 = I;
build R(V0), Q(V0), RQ(V0), Ir0 and Iq0;
compute T0 and update Ir0 and Iq0;
For k=1 , ..., n-M-1
        select j*k ;Vk = Vk-1 \ { j*k };
        update Tk, R(Vk), Q(Vk), RQ(Vk), Irk and Iqk;
end-For;
HSSB = ⌈( max_{k=0,…,n-M-1} Tk) / M⌉;
```

Figure 4.16: The HSSB bound

8.3 COMPUTATIONAL COMPLEXITY OF HSSB(I)

- Building Ir_0 and Iq_0 is done in O(nlogn) where n is the number of jobs;
- The computation of T_0 is in O(n);
- Evaluating the potential increase of T_{k-1} for each $j \in S_k$ is in constant time, hence the selection of j_k^* is in O(2M), because $|S_k| \leq 2*M$;
- The updating of T_k is in constant time;
- The updating of $R(V_{k-1})$, $Q(V_{k-1})$ and $RQ(V_{k-1})$ is in O(2M);
- The updating of Ir_k and Iq_k is in constant time;
- Hence, each loop k runs in O(4M + 2);
- There are a maximum of (n-M-1) loops, i.e. O(n).

Summing up, the complexity is O(nlogn + nM). If M > log n, as it is the case in most parallel machine problems, we can conclude that the worst running time complexity of HSSB is O(nM).

8.4 EXAMPLE

Example 4.7 shows how HSSB proceeds. Table 4.2 gives the contents of the main variables and sets of the heuristic at each step k. Note that when two jobs i and j give the same potential increase, i.e. $Incr(V_{k-1}\setminus\{i\}) = Incr(V_{k-1}\setminus\{j\})$, we have arbitrarily chosen the first evaluated job. The heuristic stops at the end of iteration 3 since $|V_3| = 3 = M+1$. After the last iteration we compute HSSB(I) as follows: $k^* = \arg \max_{k=0,\ldots,n-m-1} T_k = 3$.

$HSSB(I) = SB(V_3) = \lceil T_{k^*} / 2 \rceil = \lceil T_3/2 \rceil = \lceil 17/2 \rceil = 9$. In this example HSSB has found the optimal solution of the subset bound (SSB). $SSB(I) = SB(V^*)=9$, where $V = \{4,5,6\}$.

Example 4.7: I = {1, 2, 3, 4, 5, 6}, M = 2

j	r_j	p_j	q_j
1	1	1	5
2	2	1	1
3	1	2	1
4	3	2	3
5	4	2	2
6	3	2	3

k	j_k^*	$Incr(V_{k-1}\setminus\{ j_k^* \})$	$R(V_k)$	$Q(V_k)$	$RQ(V_k)$	T_k	V_k	Ir_k	Iq_k
0	/	/	1	2	3	14	1,2,3,4,5,6	2,4,6,5	5,4,6,1
1	**1**	$r_2-r_1-p_1= 0$	∅	∅	2, 3	14	2,3,4,5,6	4,6,5	5,4,6
2	**2**	$r_4+q_5-r_2-p_2-q_2=1$	4	5	3	15	3,4,5,6	6,5	4,6
3	**3**	$r_6+q_4-r_3-p_3-q_3=2$	6	5	4	17	4,5,6	5	4

Table 4.2: Contents of the main variables and sets of HSSB at each step k, Example 4.7

8.5 IMPROVED HSSB

REMOVING JOBS ALLOWING NON NEGATIVE INCREASE

In equation (4.18) we remove from set V_{k-1} the job j_k^* that allows the maximum increase of T_{k-1}. We can reduce the practical complexity by removing the first encountered job allowing a non negative increase. Doing so, might reduce the efficiency of HSSB for finding the optimal subset. However, numerical tests have shown that this modified version, see Figure 4.17, gives the same value for all tested instances when compared to the one presented in 4.16.

HSSB bound
$V_0 = I$;
build $R(V_0)$, $Q(V_0)$, $RQ(V_0)$, Ir_0 and Iq_0;
compute T_0 and update Ir_0 and Iq_0;
For k = 1,..., n-M-1
  if $\exists\, j \in S_k$ / $Incr(V_{k-1}\setminus\{j\}) \geq 0$ and then $j_k^* = j$
  else $j_k^* = \arg\max_{j \in S_{k-1}} Incr(V_{k-1}\setminus\{j\})$
  $V_k = V_{k-1} \setminus \{ j_k^* \}$;
  update T_k, $R(V_k)$, $Q(V_k)$, $RQ(V_k)$, Ir_k and Iq_k;
end-For;
$HSSB = \lceil (\max_{k=0,\ldots,n-M-1} T_k) / M \rceil$

Figure 4.17: HSSB improved algorithm

IMPROVEMENT STEP

Actually HSSB does not guarantee finding the optimal subset bound SSB(I). Example 4.8 shows that $HSSB(I) \leq SSB(I)$. $HSSB(I) = \lceil T_4 /2 \rceil = SB(V_4) = 19$ where $V_4 = \{3, 5, 6, 7\}$. $HSSB(I) \leq SSB(I) = SB(V^*) = 20$, where $V^* = \{2, 3, 5, 6, 7\}$. HSSB is sub-optimal because it has removed job 2, which belongs to the optimal subset solution.

Example 4.8: I = {1, 2, 3, 4, 5, 6, 7, 8}, M = 2

j	r_j	p_j	q_j
1	8	1	3
2	5	1	7
3	7	3	8
4	6	2	3
5	5	6	6
6	3	7	6
7	8	2	7
8	5	3	1

k	j_k^*	$Incr(V_{k-1}\setminus\{ j_k^* \})$	$R(V_k)$	$Q(V_k)$	$RQ(V_k)$	T_k	Ir_k	Iq_k
0	/	/	2,6	1,8	∅	37	5,8,4,3,1,7	4,5,6,2,7,3
1	**2**	-1	5,6	1,8	∅	36	8,4,3,1,7	4,5,6,7,3
2	**1**	-1	5,6	4,8	∅	35	8,4,3,7	5,6,7,3
3	**4**	1	6	8	5	36	8,3,7	6,7,3
4	**8**	2	∅	∅	6,5	38	3,7	7,3
5	**6**	-2	3	7	5	36	7	3

Table 4.3: Contents of the main variables and sets of the HSSB at each step k, Example 4.8

We hereafter present an improvement routine that reintroduces some jobs that have been discarded while processing HSSB. Let $k^* = \arg \max_{k=0,\ldots,n-m-1} T_k$ and $V_{k^*} \subseteq I$ such that $HSSB(I) = SB(V_{k^*})$.

The improvement step aims at reintroducing in V_{k^*} some of the jobs that gave a negative increase and have been discarded while processing HSSB at some iteration $t<k^*$. Remember that in the course of HSSB (see Figure 4.17), at some iteration t, if no job gives a non-negative increase we remove the one with the maximum increase, i.e. the smallest decrease.

In this improvement routine, a job is reintroduced in V_{k^*} if it leads to a direct positive increase of T_{k^*}. We hereafter present four conditions showing that reintroducing a job $j \notin V_{k^*}$ leads to a positive increase of T_{k^*}.

Let $r_{[M]}$ be the M^{th} smallest release date of the jobs in set V_{k^*}
$q_{[M]}$ be the M^{th} smallest tail of the jobs in set V_{k^*}
D be the set of jobs that gave a negative increase and have been discarded by the heuristic at iteration $t < k^*$.
$T_{k^*,j}$ be the improved numerator of $SB(V_{k^*})$ by including job $j \in D$ in the set V_{k^*}.

For any job j in D

If $(r_j \geq r_{[M]})$ and $(q_j \geq q_{[M]})$ $T_{k^*,j} = T_{k^*} + p_j$
If $(r_j \geq r_{[M]})$ and $(q_j < q_{[M]})$ and $(p_j + q_j \geq q_{[M]})$ $T_{k^*,j} = T_{k^*} + p_j + q_j - q_{[M]}$
If $(r_j < r_{[M]})$ and $(q_j \geq q_{[M]})$ and $(r_j + p_j \geq r_{[M]})$ $T_{k^*,j} = T_{k^*} + r_j + p_j - r_{[M]}$
If $(r_j < r_{[M]})$ and $(q_j < q_{[M]})$ and $(r_j + p_j + q_j \geq r_{[M]} + q_{[M]})$ $T_{k^*,j} = T_{k^*} + r_j + p_j + q_j - r_{[M]} + q_{[M]}$

The improvement step proceeds in an iterative way. It scans the jobs in D one by one and removes them from D. It reintroduces in V_k only those jobs giving a positive increase and in which case $R(V_k)$, $Q(V_k)$ and $RQ(V_k)$ are updated. The improvement step is carried out in O(nM), i.e. $|D| \leq n-M-1$ and the updating of $R(V_{k-1})$, $Q(V_{k-1})$ and $RQ(V_{k-1})$ is carried out in O(M). In Example 4.8, D = {2,1} and reintroducing job 2 allows finding the optimal subset, that is $V^*=\{2,3,5,6,7\}$.

9. NUMERICAL TESTS

All heuristics presented in this chapter were tested on several shop floor classes of configurations. Each class of configuration uses different data range (processing times, release dates and tails). Table 4.4 gives 9 classes of configurations. For a subset of classes the processing time range is maintained the same while release date and tail ranges change. These different data ranges were designed to represent different locations of the single stage in an HFS shop floor, as one of the first stages (C13, C23 and C33 where tails might be larger than release dates), one of the middle stages (C11, C21 and C31) or one of the last stages (C12, C22 and C32 where release dates might be larger than tails).

Data is generated uniformly in these ranges. For each class we tested several configurations in terms of number of jobs and number of machines (see Table 4.5). For each instance of these single stage subproblem configurations we tested 20 instances.

	C11	C12	C13	C21	C22	C23	C31	C32	C33
$p_j \in$	[10-20]	[10-20]	[10-20]	[10-30]	[10-30]	[10-30]	[10-50]	[10-50]	[10-50]
$r_j \in$	[5-15]	[5-25]	[5-15]	[5-25]	[5-55]	[5-25]	[5-55]	[5-85]	[5-85]
$q_j \in$	[5-15]	[5-15]	[5-25]	[5-25]	[5-25]	[5-55]	[5-55]	[5-25]	[5-55]

Table 4.4: Data ranges of the tested classes

n	10	10	10	20	20	20	30	30	30	50	50	50	50	80	80	80
M	2	3	4	3	4	5	3	4	5	4	5	6	7	7	8	9

Table 4.5: Configurations tested in terms of number of jobs (n) and machines (M)

Table 4.6 and 4.7 give the results obtained for each tested lower bound. Each column corresponds to one lower bound. The preemptive and partially preemptive bound (PB and PPB) were tested using the original "Preflow-push" algorithm (PB0 and PPB0) and using the new improved one (PB1 and PPB1) presented in Subsection 7.4. Table 4.6 gives the percentages that the lower bound has provided the best value, e.g. for C11, for 80% of the instances tested SB gives the best value. Table 4.7 gives the running time in seconds needed for 540 instances run under Windows NT on a Pentium II 350MHz with 64MB. The following observations based on numerical tests can be made.

A first global remark is that the larger is the data range the lower is the performance lower bounds compared to the best one, i.e. DHB.

BEST VALUE BASED ANALYSIS

- ***DHB.*** The dual heuristic based bound gives the best value, on average 99% of the best values are given by DHB. Which also means that for 1% of the tested instances DHB was dominated by another lower bound (by PPB).
- ***PPB.*** Apart from DHB, PPB outperforms all other lower bounds including PB as it has been proven in its presentation. (see Section 6). PPB gives 83% of the best values versus 77% for PB.
- ***PB.*** As it has been claimed in Vandevelde et. al., (1995), PB gives the same results as SSB.
- ***SSB, ASSB versus HSSB.*** ASSB gives the same results as SSB. Also, we can see that our proposed heuristic for the subset bound HSSB gave the same results as SSB for all tested instances.
- ***SRB, ASB versus SB***. We observed that in all test cases, SRB is dominated by SB. In half of the cases (see Table 4.5), as M divides n, SRB is worse than SB (see how RB and SRB are computed in Section 2). In the other cases we observed that SB dominated SRB because what is lost in the average machine release dates and tails is not compensated by a gain in the average machine processing times. Although potentially better, ASB gave always the same results as SB. This because the improvement in ASB with respect to SB never occurred in our randomly generated test problems.

Class	*SB*	ASB	SRB	SSB	*HSSB*	ASSB	PB0	PB1	PPB0	PPB1	*DHB*
C11	*80*	80	41	85	*85*	85	85	85	88	88	*99*
C12	*74*	74	40	83	*83*	83	83	83	88	88	*99*
C13	*72*	72	37	82	*82*	82	82	82	89	89	*99*
C21	*63*	63	31	74	*74*	74	74	74	79	79	*99*
C22	*57*	57	36	76	*76*	76	76	76	82	82	*98*
C23	*58*	58	34	76	*76*	76	76	76	84	84	*99*
C31	*53*	53	30	71	*71*	71	71	71	78	78	*99*
C32	*55*	55	35	75	*75*	75	75	75	80	80	*100*
C32	*51*	51	35	71	*71*	71	71	71	78	78	*99*
Average	***63***	***63***	***35***	***77***	***77***	***77***	***77***	***77***	***83***	***83***	***99***

Table 4.6: Percentage that a lower bound gave the best value

n	*SB*	ASB	SRB	SSB	*HSSB*	ASSB	PB0	PB1	PPB0	PPB1	*DHB*
10	*0.1*	0.3	0.1	0.5	*0.2*	0.4	1.3	1.2	113	90	*167*
20	*0.2*	0.5	0.2	1.5	*0.3*	1.0	5.7	4.2	430	343	*319*
30	*0.1*	0.7	0.2	3.8	*0.5*	1.1	11.9	8.1	744	596	*329*
50	*0.3*	1.0	0.6	11.7	*1.0*	2.1	26.1	18.6	1446	1074	*457*
80	*0.3*	2.5	1.0	41.0	*1.7*	4.5	58.9	39.9	3210	2383	*564*
Average	***0.2***	***0.9***	***0.4***	***10.1***	***0.6***	***1.5***	***17.2***	***11.9***	***957***	***719***	***301***

Table 4.7: Running time in seconds needed for 540 instances on a 350 MHz Pentium II

RUNNING TIME VERSUS BEST VALUE ANALYSIS

- For the *Preflow-push* algorithm, we can see that our improved version (used in PB1 and PPB1) runs 25-30% faster than the original one (used in PB0 and PPB0) while providing the same results.
- DHB dominates PPB, it provides better bounds and runs faster.

- Apart from DHB, PPB dominates the remaining lower bounds but is much more time consuming.
- HSSB runs faster than SSB, ASSB and PB while proving the same bounds
- HSSB is dominated by DHB but runs 300 times faster.
- SB runs faster than SRB and ASB while proving at least the same bounds.
- SB is dominated by HSSB but runs a bit faster.

HSSB always gave in the tests the same value as SSB because this simple greedy procedure, removing iteratively jobs from the set V_k (used to compute SB, see Section 8), has been constructed using necessary optimality conditions for the optimal subset V (found by SSB) when removing the job providing a positive increase in lower bound. The only case where HSSB may be suboptimal, is when HSSB removes a job providing a negative increase (see Example 4.8) in which case we used the improvement step (see Subsection 8.5).

A major or main difference between SSB and PB is that SSB allows job splitting. Indeed, in order to find a schedule of length SSB, a giving job may be processed simultaneously in parallel machines. However, it seems to be easy to avoid job splitting, which may explain why SSB gives the same value as PB. Indeed, the only way that job splitting cannot be avoided in an optimal solution of SSB is that at some time t only *one job* is ready for processing. The probability that this may happen is low with a large number of jobs because of the uniform distribution used for release dates.

PPB gives better results than PB because splitting is not allowed and preemption is more constrained. DHB gives better results than PPB because it is based on the linear relaxation of the original non-preemptive problem. The fact that this linear relaxation is often tight allows one to get a better bound than PPB.

NUMERICAL TEST CONCLUSIONS

According to these observations we retain only three of all the tested lower bounds, that is SB, HSSB and DHB. DHB gives the best lower bounds but its running time complexity is very high and therefore cannot be used in a branch and bound framework. Actually, the lower bound value given by DHB *is not far from* the one given by HSSB, and DHB does not have uniform running time, i.e. for the same instance size the running time vary. Indeed, HSSB is already a good bound, it gave the same values as PB and provided 77% of the best lower bound values. Recall that DHB uses bisection on the interval $[\delta_{LB}, \delta_{UB}]$ and has a pseudo-polynomial running time complexity. However, DHB can be used at the root node of a branch and bound algorithm to provide a good starting bound as well as for evaluating heuristic performances, because it is computed only once for each instance. HSSB seems to give the best value running time complexity ratio. In the next chapter these selected lower bounds are used in the development of branch and bound algorithms.

10. CONCLUSION

In this chapter we studied lower bounds for the makespan minimization of an HFS scheduling problem. All lower bounds are based upon the single stage parallel machine relaxation subproblem with release dates and tails. The best known lower bounds were the preemptive bound computed in $O(p_{max}n^3)$ and subset bound in $O(n^2)$ where n is the number of jobs and Pmax is the maximum processing time. Three new lower bounds have been introduced. The first lower bound is a Dual Heuristic based Bound (DHB) induced by a time indexed Linear Program (LP) formulation where the objective is to build a non-preemptive optimal schedule.

The second lower bound is obtained by constraining the preemptive assignment of jobs. It is also based on a time indexed LP formulation that produces a Partially Preemptive schedule or Bound (PPB). PPB is tighter than PB but its running time complexity is higher. PB and PPB rely on the maximum flow algorithm. We have introduced an improved version of the "Preflow-Push" algorithm for bipartite graph which runs faster.
The third lower bound is a Heuristic for SSB (HSSB) which can be calculated in a time complexity of O(nM) where M is the number of machines.

All existing lower bounds were tested on several classes of configurations. For the "*Preflow push"* algorithm our improved version runs 25-30% faster than the original one. Also, based on numerical tests, DHB was retained as the best lower bound giving the largest percentage of the best-obtained values, i.e. 99%. Apart from DHB, PPB dominates all the other tested lower bounds including PB but its running time complexity is very high. DHB dominates PPB and runs much faster. Unfortunately, DHB has a pseudo-polynomial running time complexity. For all tested instances HSSB gives the same values as PB and SSB and runs faster. Actually, HSSB gives the best value running time complexity ratio. Only three of all tested lower bounds are retained, i.e. the set bound SB, HSSB and DHB. According to our tests none dominates the others ones. These lower bounds will be used for developing branch and bound algorithms we are to study in the next chapter.

REFERENCES

1. Ahuja R. K., T. H. Magnanti, and J. B. Orlin, (1993). Network Flows. Theory, Algorithms, and Applications, Prentice Hall, New Jersey, USA.
2. Ahuja R. K., M. Kodialam, A. K. Mishra, and J. B. Orlin, (1997). Computational investigations of maximum flow algorithms, European Journal of Operational Research, 97, 509-542.
3. Carlier J., (1987). Scheduling jobs with release dates and tails on identical machines to minimize the Makespan, European Journal of Operations Research, 29, 298-306.
4. Hoogeveen J. A., J. K. Lenstra and B. Veltman, (1996). Preemptive scheduling in a two-stage multiprocessor flowshop is NP-Hard, European Journal of Operations Research 89(22), 172-175, 1996.

5. Goldberg A.V., and R.E. Tarjan (1986), 'A new approach to the maximum flow problem', Proceedings of the 18th ACM Symposium on the Theory of Computing, 136-146. Full paper in Journal of ACM 35 (1988), 921-940.
6. Horn W.A., (1974). Some simple scheduling algorithms, Naval Research Logistics Quarterly 21, 177-185.
7. Labetoulle J., E.L. Lawler, J. K. Lenstra, and A.H.G. Rinnooy Kan, (1984). Preemptive scheduling of uniform machines subject to release dates, Progress in Combinatorial Optimization, 245-261, Academic Press, Florida.
8. Sousa J.P., (1989). Time indexed formulations of non-preemptive single-machine scheduling problems. Ph.D. thesis. Université Catholique de Louvain, Facultés des Sciences Appliquées, Louvain-La-Neuve, Belgium.
9. Sousa J.P. and L.A. Wolsey, (1992). A time indexed formulation of non-preemptive single-machine scheduling problems, Mathematical programming, 54:(3), 353-367, 16, 1992.
10. Vandevelde A., H. Hoogeveen, C. Hurkens and J. K. Lenstra, (1995). Lower bounds for the multiprocessor flowshop. Working paper, Eindhoven University of Technology, The Netherlands.