OPTIMIZATION OF THE FLOWSHOP SCHEDULING PROBLEM WITH SETUP TIMES

by

ROGER ZIRAHUÉN RíOS MERCADO, Lic., M.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 1997

OPTIMIZATION OF THE FLOWSHOP SCHEDULING PROBLEM WITH SETUP TIMES

APPROVED BY
DISSERTATION COMMITTEE:

Supervisor: _____

_____

_____

_____

_____

To my wife Ofelia,

to my son Vandari,

to my parents Griselda and Rogelio,

to my siblings Azucena, Netza, and Fidel,

to all my relatives,

and to all those I deeply love and care about.

Acknowledgements

I am deeply grateful to my dissertation supervisor, Prof. Jonathan F. Bard, for his guidance and encouragement throughout this work, and for taking our academic relationship to a more personal level.

I am grateful also to the members of my dissertation committee, Paul Jensen, Leon Lasdon, Gang Yu, and David Morton, for their positive comments, insights, and suggestions, that greatly improved the quality of this research.

I also want to express my sincere gratitude to Matthew Saltzman, Matteo Fischetti, Paolo Toth, Manfred Padberg, and Giovanni Rinaldi, for kindly allowing me to use their computer codes, and to all those fellow researchers and people who have, in one way or another, contributed to this research.

I thank each of the faculty members (current and former) and fellow graduate students of the Graduate Program in Operations Research and the Graduate Program in Management Science and Information Systems at the University of Texas at Austin, for all those unforgettable moments we had both in and outside the classroom.

I am profoundly grateful to José Luis González Velarde, who played a very important role on my decision of coming to UT-Austin.

Special thanks to our dearest friends in Austin, who made our lives and our stay in this city such a joyful, cultural-rich, and truly exceptional experience. I will never, ever, forget all those very special and exciting moments my family and I shared with them.

Finally, my research work would have not been possible without the following financial support: a grant for doctoral studies from the Mexican National Council of Science and Technology (CONACYT), and an E. D. Farmer Fellowship, a Continuing Fellowship, and a David Bruton Fellowship, all from the University of Texas at Austin.

OPTIMIZATION OF THE FLOWSHOP SCHEDULING PROBLEM WITH SETUP TIMES

Publication No. _____

Roger Zirahuén Ríos Mercado, Ph.D.
The University of Texas at Austin, 1997

Supervisor: Jonathan F. Bard

This dissertation addresses the optimization of the flowshop scheduling problem with sequence-dependent setup times. The goal of the decision-maker is to provide a schedule that minimizes the time at which all jobs in the system are processed. The goal of this work is to provide the decision-maker with efficient ways of deriving such schedules. This type of problem arises in many manufacturing environments. In the printing industry, for example, presses must be cleaned and settings changed when ink color, paper size or receiving medium differ from one job to the next. Setup times are strongly dependent on the job order. In the container manufacturing industry machines must be adjusted whenever the dimensions of the containers are changed, while in printed circuit board assembly, rearranging and restocking component inventories on the magazine rack is required between batches. In each of these situations, sequence-dependent setup times play a major role and must be considered explicitly when modeling the problem.

This research includes the implementation of heuristic approaches to obtain feasible solutions of high quality, the development of lower bounding

procedures, the study of the set of feasible solutions from the polyhedral perspective, and the integration of all of these components into exact optimization schemes. The first is based on branch and cut and the second on branch and bound. The proposed procedures are found to be very effective, providing good approximations of the true optimal to a large class of data instances, and optimal solutions in other cases.

Another contribution of this work is the development of a technique to randomly generate data instances with real-world attributes.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this dissertation, we address the problem of finding a permutation schedule of n jobs in an m-machine flowshop environment that minimizes the maximum completion time $C_{\max}$ of all jobs, also known as the makespan. The jobs are available at time zero and have sequence-dependent setup times on each machine. All parameters, such as processing and setup times, are assumed to be known with certainty. This problem is regarded in the scheduling literature as the sequence-dependent setup time flowshop (SDST flowshop or $F|s_{ijk},\ prmu|C_{\max}$ using the scheduling notation described in Appendix A) and is evidently $\mathcal{NP}$-hard since the case where $m = 1$ is simply a traveling salesman problem (TSP).

Applications of sequence-dependent scheduling are commonly found in most manufacturing environments. In the printing industry, for example, presses must be cleaned and settings changed when ink color, paper size or receiving medium differ from one job to the next. Setup times are strongly dependent on the job order. In the container manufacturing industry machines must be adjusted whenever the dimensions of the containers are changed, while in printed circuit board assembly [46], rearranging and restocking component inventories on the magazine rack is required between batches. In the chemical and pharmaceutical industry [5], the processing of different chemical compounds in a specific machine may require some cleansing between process runs,

and the time to set up a machine for the next task may be strongly dependent on the immediate predecessor. In the problem of scheduling an aircraft approaching or leaving a terminal area [16], the time separations between successive aircraft belonging to different categories are to be changed according to their respective position. Thus sequence-dependent processing times must be allowed for a more realistic description of the problem. brand will mingle with the scent of the following brand unless the machine is cleaned thoroughly and very carefully resulting in high setup times. In each of these situations, sequence-dependent setup times play a major role and must be considered explicitly when modeling the problem.

The objective of this work is to provide effective methods to find exact or high quality approximate solutions to this problem. This includes the implementation of heuristics to obtain good feasible solutions, the development of lower bounding procedures, the study of the set of feasible solutions from a polyhedral theory perspective, and the integration of all of these elements into exact optimization schemes.

As far as approximate solutions are concerned, we present an hybrid heuristic that exploits the underlying asymmetric traveling salesman problem (ATSP) and a greedy randomized adaptive search procedure (GRASP). GRASP is a heuristic approach to combinatorial optimization problems that combines greedy heuristics, randomization, and local search techniques. Both heuristics are found to be very effective on finding feasible solutions of high quality.

Recent developments on the polyhedral structure of the ATSP and the similarities between the ATSP and the SDST flowshop motivated our study on the SDST flowshop polyhedron; i.e., the convex hull of incidence vectors of all feasible solutions. In so doing, we consider two different models or formulations. Model A is based on the asymmetric traveling salesman problem (ATSP) and

model B is based on a formulation due to Srikar and Ghosh [66]. In each case, two sets of variables are identified: a set of binary decision variables which determines the sequence or ordering of the jobs, and a set of nonnegative real variables which determines the times processing begins for each job. When the time variables are ignored the binary variables give rise to a subspace of the SDST flowshop consisting of the convex hull of incidence vectors of feasible sequences. For model A, this subspace is the well known ATSP polytope; for model B, the corresponding subspace (here, called the S-G polytope) has not been previously studied. In our work, we show how any facet-defining inequality (or facet) for either of these polytopes induces a facet for the SDST flowshop polyhedron. We also investigate the facial structure of the S-G polytope and develop several valid inequalities for the SDST flowshop polyhedron. We find these valid inequalities to be effective when incorporated into a branch-and-cut (B&C) exact optimization algorithm; however, this effectiveness was somehow limited by the fact that the linear programming lower bound of the relaxed subproblems was still not tight enough.

By relaxing some machine requirements rather than the integrality conditions on the MIP formulations, alternate lower bounding procedures were developed. The generalized lower bound (GLB) is obtained by reducing the original m-machine to a 2-machine problem, and the machine-based lower bound (MBLB) is obtained by reducing to a single machine problem. Both procedures were found to be marginally better than the LP-relaxation lower bound, with the MBLB being more effective than the GLB.

Extending these lower bounding procedures to handle partial schedules as well, enabled us to develop a branch-and-bound enumeration scheme. This scheme included, in addition to the lower and upper bounding procedures, a dominance elimination criterion and several searching strategies to provide us with an intelligent way of searching for optimal solutions. This algorithm was

found extremely effective, providing optimal solutions to a large class of data instances, and near-optimal solutions in other cases.

The dissertation is organized as follows. The most relevant work on this area is presented in Chapter 2. In Chapter 3, we introduce the mathematical models A and B, and discuss their basic differences and properties. Major results relating the polyhedral structure of models A and B are given in Chapter 4. Then, the effectiveness of the valid inequalities within a B&C framework is assessed in Chapter 5 which includes a discussion of separation algorithms, and a B&C code implementation. The heuristic procedures are described and evaluated in Chapter 6. In Chapter 7, we present a full description and extensive computational experimentation of the branch-and-bound enumeration algorithm, including a discussion of non-LP-based lower bounds and dominance rules. We conclude with a discussion of the results and directions for future research in Chapter 8.

Chapter 2

Related Work

In this section we highlight the main contributions to the flowshop scheduling field. For a description of the notation, see Appendix A. Lawler et al. [44] present an extensive survey, concentrating on the area of deterministic machine scheduling. They review complexity results and optimization and approximation algorithms involving a single machine, parallel machines, open shops, flow shops and job shops. They also pay attention to two extensions of this area: resource-constrained project scheduling and stochastic machine scheduling.

Blazewicz et al. [7] present a review of a variety of deterministic machine scheduling problems. They overview the existing results and present solution strategies for resource-constrained scheduling, scheduling tasks that require more than one machine at a time, scheduling with nonlinear speed-resource alloted functions, and scheduling in flexible manufacturing systems.

Blazewicz et al. [6] present a survey that compiles a large number of mathematical programming formulations for a variety of machine scheduling problems. Their formulations include single machine scheduling, parallel machine nonpreemptive scheduling, parallel machine preemptive scheduling, job shop scheduling, and parallel machine scheduling with nonlinear speed-resource amount functions.

2.1 Minimizing Makespan on Regular Flowshops

The flowshop scheduling problem (with no setups) has been studied extensively over the past 25 years. The pioneering work in flowshop scheduling dates back to 1954 when Johnson [39] presented a simple decision rule for solving $F2||C_{\max}$ to optimality in polynomial time. He also discussed how to solve a special case of $F3||C_{\max}$. Nevertheless, virtually all other cases of the flowshop problem are hard problems.

2.1.1 Exact Optimization Schemes

Several exact optimization schemes, mostly based on branch and bound, have been proposed for $F||C_{\max}$ including those of Lageweg et al. [42], Potts [58] and Carlier and Rebai [10]. The 3-machine special case of this problem is considered by Ignall and Schrage [38] and Lomnicki [47]. Della Croce et al. [17] present a branch-and-bound approach for the 2-machine case.

2.1.2 Heuristics

Heuristic approaches for $F||C_{\max}$ can be divided into (a) quick procedures [56, 9, 28, 15, 68, 51, 35, 34, 63, 49] and (b) extensive search procedures [75, 71, 54] (including techniques such as tabu search). Several studies have shown (e.g., [72]) that the most effective quick procedure is the heuristic due to Nawaz et al. [51]. In our work, we attempt to take advantage of this result and extend their algorithm to the case where setup times are included within a randomized algorithm. Our implementation, GRASP, is further described in Section 6.3.

2.2 Sequence-Dependent Setup Times

2.2.1 Exact Optimization Schemes

Multiple-machine case: The best efforts to solve the problem optimally have been made by Srikar and Ghosh [66], and later by Stafford and Tseng [67] in terms of solving MIP formulations. Srikar and Ghosh introduce a formulation that uses half the number of binary variables in the TSP-based formulation. They used this model and the SCICONIC/VM mixed integer programming solver (based on branch and bound) to solve several randomly generated instances of the SDST flowshop. The largest solved was a 6-machine, 6-job problem, in about 22 minutes of CPU on a Prime 550 computer.

Later, Stafford and Tseng corrected an error in the S-G formulation and using LINDO solved a 5×7 instance in about 6 hours of CPU time on a PC. They also proposed three new MIP formulations of related flowshop problems based on the S-G model.

To the best of our knowledge, there have been no better approaches to solve the problem optimally. However, Gupta [32] presents a branch-and-bound algorithm for the case where the objective is to minimize the total machine setup time. No computational results are reported.

Two-machine case: Work on $F2|s_{ijk}, prmu|C_{\max}$ includes Corwin and Esogbue [12], who consider a subclass of this problem that arises when one of the machines has no setup times. After establishing the optimality of permutation schedules, they develop an efficient dynamic programming formulation which they show is comparable, from a computational standpoint, to the corresponding formulation of the traveling salesman problem. No algorithm is developed.

Gupta [29] establishes some complexity results for special cases. After showing the $\mathcal{NP}$-hardness of this problem he proposes a TSP formulation for the case where jobs are processed continuously through the shop. He uses

these results to describe an approximate algorithm for the case where limited or infinite intermediate storage space is available to hold partially completed jobs.

Gupta and Darrow [30] establish the $\mathcal{NP}$-hardness of the problem and show that permutation schedules do not always minimize makespan. They derive sufficient conditions for a permutation schedule to be optimal, and propose and evaluate empirically four heuristics. They observe that the procedures perform quite well for problems where setup times are an order of magnitude smaller than the processing times. However, when the magnitude of the setup times was in the same range as the processing times, the performance of the first two proposed algorithms decreased sharply.

Szwarc and Gupta [69] develop a polynomially bounded approximate method for the special case where the sequence-dependent setup times are additive. Their computational experiments show optimal results for the 2-machine case.

1-Machine Case: Work on the single-machine case includes Lockett and Muhlemann [46], who address a special case where the setup takes the form of number of tool changes from one job to the next. The jobs differ considerably in their tool requirements and may need tools that are not presently on the turret. The authors focus on minimizing the total number of tool changes. This problems differs from the typical sequence-dependent setup time problem in that the changeover for the next job depends not only on the previous job, but on all preceding jobs, if some of the stations are empty. They describe several heuristics based on the traveling salesman problem and report that the TSP heuristic without backtracking gives the best results. They also derive a branch-and-bound procedure, but it was found to be ineffective in all but the smallest instances.

White and Wilson [74] developed a procedure that classifies setup operations and predicts the setup times for a single-machine problem. Then, based on these predictions, they develop a heuristic based on the TSP to sequence the jobs. No computational experience is reported.

Bianco et al. [5] propose a branch-and-bound algorithm for $1|r_j, s_{jk}|C_{\max}$. After establishing some lower and upper bounding schemes and dominance criteria rules, they test their algorithm on 10-, 15- and 20-job instances with processing times randomly generated in $[1, 10]$ and ready times randomly distributed in the following three intervals $[0, 25]$, $[0, 40]$, and $[0, 50]$ for the 10 and 15-job instances; and in $[0, 100]$ for the 20-job instances. For the $[0, 25]$ interval, the average CPU time for over 50 randomly generated instances was 4.2 sec., and 26.2 sec. for the 10 and 15 job instances, respectively. For the $[0, 50]$ interval, average CPU time was 20.6 sec. and 409.5 sec. for the 10 and 15 job instances. The 20-job instances took, in average, 115.3 sec. of CPU time.

Gupta et al. [31] considered the single machine bi-criteria scheduling problem where jobs are from multiple classes and where customer orders consist of at least one job from each of the classes. A setup time is needed whenever the machine is changed over from a job in one class to a job in another. One objective is to minimize the makespan which is equivalent to minimize the total setup time. The other objective is to minimize total carrying costs of the customer orders. This cost is measured by the length of the time interval between the completion times of the first job and the last job in the customer order. They proposed polynomial-time algorithms for the two bi-criteria scheduling problems in which one objective is to be optimized while holding the other objective fixed at its optimal value.

Van der Veen and Zhang [73] considered a problem where n jobs are to be processed on a single machine such that the total required change-over time

is minimized. They further assumed that the jobs can be divided into K classes and that the change-over time between two consecutively scheduled jobs solely depends on the job-classes that the two jobs belong to. They showed that this problem is solvable in $O(n)$ time for K fixed.

Zdrzałka [76] considered the single-machine scheduling problem in which each job has a release date, a delivery time, and a sequence-independent setup time. Preemption is permitted and the objective is to minimize the time by which all jobs are delivered. He showed that the problem is $\mathcal{NP}$-hard and proposed an $O(n \log n)$ heuristic with a tight worst-case performance bound of $\frac{3}{2}$.

Hansmann [33] presented several heuristics for the setup cost minimization in a problem arising from a major cigarette company in Germany. His threshold heuristic gave better results than his simulated annealing heuristic in problems with up to 120 jobs.

2.2.2 Heuristics

The most relevant work on heuristics for $F|s_{ijk},\ prmu|C_{\max}$ is due to Simons [65]. He describes four heuristics and compares them with three benchmarks that represent generally practiced approaches to scheduling in this environment. Experimental results for problems with up to 15 machines and 15 jobs are presented. His findings indicate that two of the proposed heuristics (`SETUP` and `TOTAL`) produce substantially better results than the other methods tested. In this work, we provide an enhanced version of these heuristics by correcting some of its shortcomings and by adding a local search phase. Full description is given in Section 6.2.

Chapter 3

Mathematical Formulation

3.1 Statement of Problem

In the flowshop environment, a set of n jobs must be scheduled on a set of m machines, where each job has the same routing. Therefore, without loss of generality, we assume that the machines are ordered according to how they are visited by each job. Although for a general flowshop the job sequence may not be the same for every machine, here we assume a *permutation schedule*; i.e., a subset of the feasible schedules that requires the same job sequence on every machine. We suppose that each job is available at time zero and has no due date. We also assume that there is a setup time which is sequence dependent so that for every machine i there is a setup time that must precede the start of a given task that depends on both the job to be processed (k) and the job that immediately precedes it (j). The setup time on machine i is denoted by s_{ijk} and is assumed to be *asymmetric*; i.e., $s_{ijk} \neq s_{ikj}$. After the last job has been processed on a given machine, the machine is brought back to an acceptable "ending" state. We assume that this last operation takes zero time because we are interested in job completion time rather than machine completion time. Our objective is to minimize the time at which the last job in the sequence finishes processing on the last machine, also known as *makespan*. This problem is denoted by $Fm|s_{ijk},\ prmu|C_{\max}$ or SDST flowshop.

In modeling this problem as a mixed-integer program (MIP), we consider

two different formulations. In the first case, a set of the binary variables is used to define whether or not one job is an immediate predecessor of another; in the second case, the binary variables simply determine whether or not one job precedes another. A set of nonnegative real variables is also included in the formulations. In either case, they have the same definition and are used to determine the starting time of each job on each machine.

Example 3.1 Consider the following instance of $F2|s_{ijk},\ prmu|C_{\max}$ with four jobs.

p_{ij}	1	2	3	4
1	6	3	2	1
2	2	2	4	2

s_{1jk}	1	2	3	4
0	3	4	1	7
1	-	5	3	2
2	5	-	3	1
3	2	1	-	5
4	3	2	5	-

s_{2jk}	1	2	3	4
0	2	3	1	6
1	-	1	3	5
2	4	-	3	1
3	3	4	-	1
4	7	8	4	-

A schedule $S = (3, 1, 2, 4)$ is shown in Figure 3.1. The corresponding makespan is 24, which is optimal. □

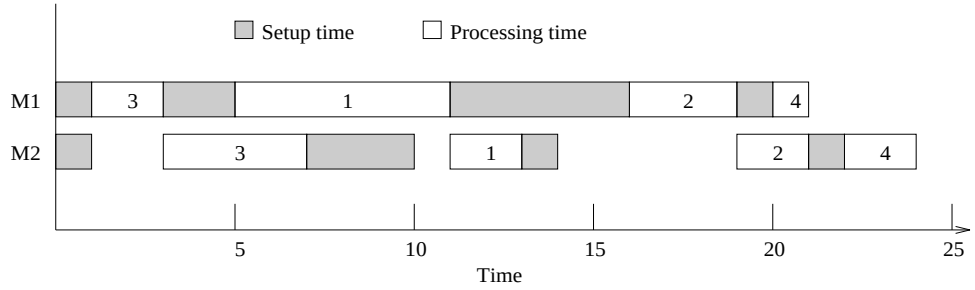


Figure 3.1: Example of a 2×4 SDST flowshop

Triangle inequality: The triangle inequality for the setup times is stated as follows:

$$s_{ijk} + s_{ikl} \geq s_{ijl} \qquad i \in I,\ j, k, l \in J. \tag{3.1}$$

Throughout the sequel, we will assume that the triangle inequality holds. In most operations (e.g., see [66, 67]), the time it takes to set up a machine from job j to job l is less than the time it takes to set up a machine from j to another job k, and then set up the machine from k to l. Nevertheless, if there really exists a machine i and jobs j, k, l such that $s_{ijk} + s_{ikl} < s_{ijl}$, we can always replace s_{ijl} with $s'_{ijl} = s_{ijk} + s_{ikl}$ and force (3.1) to hold as an equality.

3.2 Notation

In the development of the mathematical programming model, we make use of the following notation.

Indices and sets

m	number of machines
n	number of jobs
i	machine index; $i \in I = \{1, 2, \ldots, m\}$
j, k, l	job indices; $j, k, l \in J = \{1, 2, \ldots, n\}$
J_0	$= J \cup \{0\}$ extended set of jobs, including a dummy job denoted by 0

Input data

p_{ij}	processing time of job j on machine i; $i \in I$, $j \in J$
s_{ijk}	setup time on machine i when job j is scheduled right before job k; $i \in I$, $j \in J_0$, $k \in J$

Computed parameters

A_i upper bound on the time at which machine i finishes processing its last job; $i \in I$,

$$A_i = A_{i-1} + \sum_{j \in J} p_{ij} + \min \left\{ \sum_{j \in J_0} \max_{k \in J} \{s_{ijk}\}, \sum_{k \in J} \max_{j \in J_0} \{s_{ijk}\} \right\}$$

where $A_0 = 0$

B_{ij} lower bound on the starting time of job j on machine i; $i \in I$, $j \in J$

$$B_{ij} = \max \{s_{i0j}, B_{i-1,j} + p_{i-1,j}\} \qquad i \in I, \ j \in J$$

where $B_{0j} = 0$ for all $j \in J$

Common variables

y_{ij} nonnegative real variable equal to the starting time of job j on machine i; $i \in I$, $j \in J$

$C_{\max}$ nonnegative real variable equal to the makespan;

$$C_{\max} = \max_{j \in J} \{y_{mj} + p_{mj}\}$$

3.3 Model A

Let $A = \{(j,k) : j,k \in J_0, \ j \neq k\}$ be the set of arcs in a complete directed graph induced by the node set J_0. We define the decision variables as follows:

$$x_{jk} = \begin{cases} 1 & \text{if job } j \text{ is the immediate predecessor of job } k; \ (j,k) \in A \\ 0 & \text{otherwise} \end{cases}$$

In the definition of x_{jk}, notice that $x_{0j} = 1$ ($x_{j0} = 1$) implies that job j is the first (last) job in the sequence for $j \in J$. Also notice that s_{i0k} denotes the initial setup time on machine i when job k has no predecessor; that is, when job k is scheduled first, for $k \in J$. This variable definition yields what we call a TSP-based formulation.

$$\text{Minimize} \quad C_{\max} \tag{3.2.1}$$

subject to

$$\sum_{j \in J_0} x_{jk} = 1 \qquad k \in J_0 \tag{3.2.2}$$

$$\sum_{k \in J_0} x_{jk} = 1 \qquad j \in J_0 \tag{3.2.3}$$

$$y_{ij} + p_{ij} + s_{ijk} \leq y_{ik} + A_i(1 \Leftrightarrow x_{jk}) \qquad i \in I,\ j,k \in J \tag{3.2.4}$$

$$y_{mj} + p_{mj} \leq C_{\max} \qquad j \in J \tag{3.2.5}$$

$$y_{ij} + p_{ij} \leq y_{i+1,j} \qquad i \in I \setminus \{m\},\ j \in J \tag{3.2.6}$$

$$x_{jk} \in \{0,1\} \qquad (j,k) \in A \tag{3.2.7}$$

$$y_{ij} \geq B_{ij} \qquad i \in I,\ j \in J \tag{3.2.8}$$

Equations (3.2.2) and (3.2.3) state that every job must have a predecessor and successor, respectively. Note that one of these $2n + 2$ assignment constraints is redundant in the description of the feasible set. Time-based subtour elimination constraints are given by (3.2.4), and establish that if job j precedes job k, then the starting time of job k on machine i must not exceed the completion time of job j on machine i $(y_{ij} + p_{ij})$ plus the corresponding setup time. Here, A_i is a large enough number (an upper bound on the completion time on machine i). Constraint (3.2.5) assures that the makespan is greater than or equal to the completion time of all jobs on the last machine, while (3.2.6) states that a job cannot start processing on one machine if it has not finished processing on the previous one. A lower bound on the starting time for each job on each machine is set in (3.2.8).

In formulation (3.2.1)-(3.2.8), we assume that s_{ij0}, the time required to bring machine i to an acceptable end state when job j is processed last, is zero for all $i \in I$. Thus the makespan is governed by the completion times of the jobs only. We are also assuming that all jobs need processing on all machines.

If this last condition were not true, then eq. (3.2.5) could be replaced by

$$y_{ij} + p_{ij} \quad \leq \quad C_{\max} \qquad i \in I,\ j \in J$$

at the expense of increasing the number of makespan constraints from n to mn. Note that it is possible to combine $p_{ij} + s_{ijk}$ in (3.2.4) into a single term $t_{ijk} = p_{ij} + s_{ijk}$, but that we still need to handle the processing times p_{ij} separately in constraints (3.2.5) and (3.2.6).

If the triangle inequality does not hold, constraint (3.2.8) must be replaced by

$$B_{ij} \quad \leq \quad y_{ij} + C_i(1 \Leftrightarrow x_{0j}) \qquad i \in I,\ j \in J,$$

where C_i is a large enough number (an upper bound on the initial setup time for machine i).

3.4 Model B

Srikar and Ghosh (S-G) [66] proposed a second MIP formulation for the SDST flowshop. Their formulation contained a slight error that was later corrected by Stafford and Tseng [67]. The Srikar-Ghosh model does not consider the initial setup time s_{i0k} for the first job in the sequence, that is, it is assumed to be zero. Our formulation includes this parameter.

Let $\hat{A} = \{(j,k) : j,k \in J,\ j < k\}$. The decision variables are defined as follows:

$$x_{jk} \quad = \quad \begin{cases} 1 & \text{if job } j \text{ is scheduled any time before job } k;\ (j,k) \in \hat{A} \\ 0 & \text{otherwise} \end{cases}$$

The MIP formulation is

$$\text{Minimize} \quad C_{\max} \tag{3.3.1}$$

subject to

$$
\begin{aligned}
y_{ij} + p_{ij} + s_{ijk} &\leq y_{ik} + A_i(1 \Leftrightarrow x_{jk}) && i \in I, (j,k) \in \hat{A} && (3.3.2)\\
y_{ik} + p_{ik} + s_{ikj} &\leq y_{ij} + A_i(x_{jk}) && i \in I, (j,k) \in \hat{A} && (3.3.3)\\
y_{mj} + p_{mj} &\leq C_{\max} && j \in J && (3.3.4)\\
y_{ij} + p_{ij} &\leq y_{i+1,j} && i \in I \setminus \{m\}, && \\
& && && j \in J && (3.3.5)\\
x_{jk} &\in \{0,1\} && (j,k) \in \hat{A} && (3.3.6)\\
y_{ij} &\geq B_{ij} && i \in I,\ j \in J && (3.3.7)
\end{aligned}
$$

Constraints (3.3.2) and (3.3.3) ensure that time precedence is not violated. They also eliminate cycles. Equation (3.3.4) establishes the makespan criterion. Equation (3.3.5) states that a job cannot start processing on one machine if it has not finished processing on the previous machine. A lower bound on the starting time of each job on each machine is set in (3.3.7).

Srikar and Ghosh point out that the triangle inequality must hold in order for constraints (3.3.2)-(3.3.3) to hold. However, Stafford and Tseng provide a stronger condition for constraints (3.3.2)-(3.3.3) to be valid; i.e.,

$$s_{ijk} + s_{ikl} + p_{ik} \geq s_{ijl} \qquad i \in I,\ j,k,l \in J. \tag{3.4}$$

Note that (3.4) is stronger than the triangle inequality (3.1), and implies that constraints (3.3.2)-(3.3.3) of the model hold, even if (3.1) does not hold for setup times. They illustrate this by means of an example.

If the triangle inequality does not hold, constraints (3.3.2), (3.3.3) and (3.3.7) are no longer valid. One possible replacement is

$$
\begin{aligned}
y_{ij} + p_{ij} + s_{ijk} &\leq y_{ik} + (n+1)A_i(1 \Leftrightarrow x_{jk}) + A_i[P(k) \Leftrightarrow P(j) \Leftrightarrow 1]\\
y_{ij} + p_{ij} + s_{ijk} &\leq y_{ik} + (n+1)A_i x_{jk} + A_i[P(j) \Leftrightarrow P(k) \Leftrightarrow 1]
\end{aligned}
$$

for $i \in I$, $(j,k) \in \hat{A}$ and

$$B_{ik} \leq y_{ik} + C_i[P(k) \Leftrightarrow 1]$$

for $i \in I$, $k \in J$, respectively, where C_i is a large enough number (upper bound on the starting processing time of all jobs on machine i), and $P(j)$ represents the position in the schedule of job j given by

$$P(j) \;=\; \sum_{p<j} x_{pj} + \sum_{q>j} (1 \Leftrightarrow x_{jq}) + 1 \quad j \in J. \tag{3.5}$$

In addition, the following constraints must be added to the formulation:

$$\begin{aligned} P(j) + 1 &\leq P(k) + n(1 \Leftrightarrow x_{jk}) \quad (j,k) \in \hat{A} \\ P(k) + 1 &\leq P(j) + n x_{jk} \qquad\quad (j,k) \in \hat{A} \end{aligned}$$

Thus, when the triangle inequality does not hold, the problem size increases considerably.

3.5 Model Comparison

	Model A		Model B	
Variables	Binary	$n(n+1)$	Binary	$\frac{1}{2}n(n-1)$
	Real	$mn+1$	Real	$mn+1$
	Total	$n(n+1)+mn+1$	Total	$\frac{1}{2}n(n-1)+mn+1$
Constraints	(3.2.2)	$n+1$	(3.3.2)	$\frac{1}{2}mn(n-1)$
	(3.2.3)	$n+1$	(3.3.3)	$\frac{1}{2}mn(n-1)$
	(3.2.4)	$mn(n-1)$		
	(3.2.5)	mn	(3.3.4)	mn
	(3.2.6)	$n(m-1)$	(3.3.5)	$n(m-1)$
	Total	$mn^2+mn+n+2$	Total	mn^2+mn-n
Nonzeros	(3.2.2)	$n(n+1)$	(3.3.2)	$\frac{3}{2}mn(n-1)$
	(3.2.3)	$n(n+1)$	(3.3.3)	$\frac{3}{2}mn(n-1)$
	(3.2.4)	$3mn(n-1)$		
	(3.2.5)	$2mn$	(3.3.4)	$2mn$
	(3.2.6)	$2n(m-1)$	(3.3.5)	$2n(m-1)$
	Total	$3mn^2+2n^2+mn$	Total	$3mn^2+mn-2n$

Table 3.1: Problem size for models A and B

Table 3.1 shows the problem size in terms of number of variables, constraints, and nonzeros for either model. As can be seen, model B is considerably

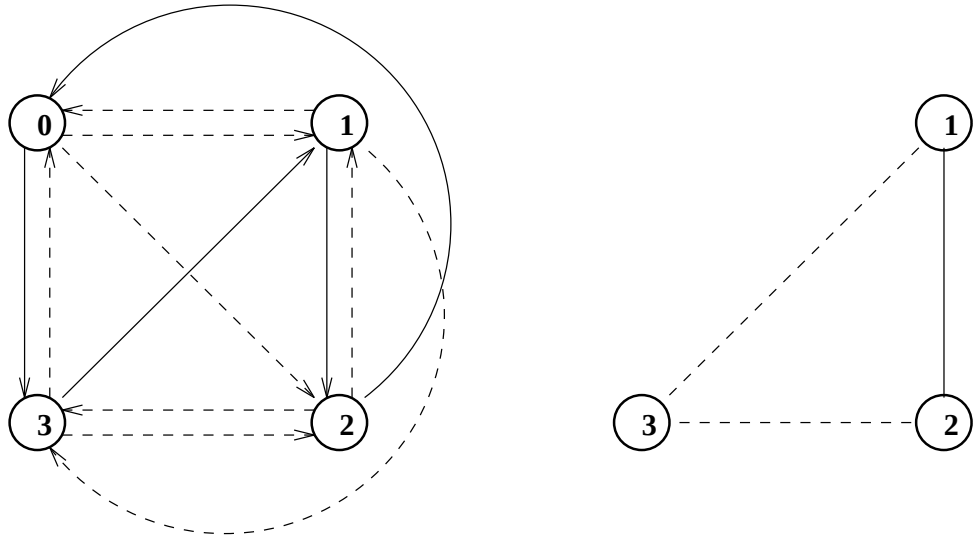
smaller than model A in terms of both the number of constraints and the number of binary variables. This would appear to make it more attractive when considering exact enumeration methods such as branch and bound (B&B) and branch and cut (B&C). Nevertheless, the fact that much is known about the ATSP polytope gives added weight to model A. Table 3.2 displays the number of binary and real variables, number of constraints, number of nonzeros and density of the matrix of constraints for several values of m and n.

$m \times n$	Model	Binary	Real	Constraints	Nonzeros	Density
2×10	A	110	21	252	840	0.025
	B	45	21	230	620	0.041
2×20	A	420	41	902	3280	0.008
	B	190	41	860	2440	0.012
10×10	A	110	101	1212	3400	0.013
	B	45	101	1190	3180	0.018
10×20	A	420	201	4422	13200	0.005
	B	190	201	4380	12360	0.007

Table 3.2: Problem size examples for models A and B

To date, it has not been possible to tackle even moderate size instances of the SDST flowshop with either of these formulations due mainly to the weakness of their LP-relaxation lower bounds. LP-based enumeration procedures such as B&B and B&C require good LP-relaxation lower bounds. For example, Stafford and Tseng required about 6 hours of CPU time on a 80286-based PC to optimally solve a 5×7 instance using LINDO with formulation B. To improve the polyhedral representation of the relaxed feasible regions it is necessary to generate valid inequalities, the strongest being facets. One way to achieve this is by looking into the related subspaces: the ATSP polytope and the S-G polytope for models A and B, respectively. Many facets have been developed for the ATSP polytope over the last 20 years (e.g., see [1, 2, 3, 24, 59]). For model B, though, the S-G polytope remains unexplored. As we show presently,

the facets of either of these polytopes can be extended to facets of the SDST flowshop polyhedron.



(a) Model A (b) Model B

Figure 3.2: Graph representations for schedule (3,1,2)

When comparing the ATSP polytope with the S-G polytope fundamental differences can be observed. In the former, we have a clear picture of what a feasible solution (also called a tour) looks like in a graph. This makes it easier to visualize, for instance, when certain constraints, such as the subtour eliminate constraints, may be violated. However, for model B, it is not a straightforward matter to identify in a graph a feasible solution from a given set of arcs. Figure 3.2 shows the graph for a 3-job problem and the solution for schedule $S = (3, 1, 2)$ for both models. For model B, an undirected graph can be used because x_{jk} is only defined for $j < k$. The dotted lines represent all feasible arcs (12 for model A and 3 for B); the solid lines identify the solution.

Figure 3.3 shows how a solution for model B can be built from a solution for model A. Note that each arc $\hat{e} \in \hat{A}$ (Step 2) is visited just once so the procedure is $O(|\hat{A}|) = O(n^2)$. In Step 1, a node (job) within brackets ($[j]$) denotes the job scheduled in the j-th position.

Procedure `A-to-B()`

Input: An arc set T (or directed tour, $|T| = n + 1$) corresponding to a feasible schedule for the SDST flowshop under model A.

Output: An arc set $\hat{T}$ corresponding to the equivalent schedule under model B.

0: Initialize `visited(`$\hat{e}$`)=NO` for $\hat{e} \in \hat{A}$ and set $\hat{T} = \emptyset$

1: Sort T as

$$T = \{(0,[1]),([1],[2]),\ldots,([n \Leftrightarrow 1],[n]),([n],0)\}$$

2: for $j = 1$ to n do

2a: Choose the j-th arc in T

2b: for each unvisited arc $\hat{e} \in \hat{A}$ incident to $[j]$ do

2c: `visited(`$\hat{e}$`)=YES`

2d: for each unvisited arc $\hat{e} \in \hat{A}$ incident from $[j]$ do

2e: `visited(`$\hat{e}$`)=YES`

2f: $\hat{T} \leftarrow \hat{T} + \hat{e}$

3: Output $\hat{T}$

Figure 3.3: Procedure to go from solution of A to solution of B

Likewise, a solution for model A can be easily constructed from a feasible solution for model B. Let $\hat{T}$ be an arc set representing a feasible schedule under model B. Let $\hat{x} \in B^{|\hat{A}|}$ be its corresponding characteristic vector; that is, $x_{jk} = 1$ if $(j,k) \in \hat{T}$, and $x_{jk} = 0$ otherwise. For each job j, its position in the schedule $P(j)$ is determined by eq. (3.5) in $n(n \Leftrightarrow 1)$ operations. The schedule S is found by sorting the jobs by increasing value of $P(j)$ and a feasible tour T is easily built from S in $O(n)$ time so that the complete conversion takes $O(n^2)$ time.

Chapter 4

Polyhedral Theory

Concepts and definitions relating basic polyhedral theory are contained in Appendix B.

4.1 Polyhedral Results for Model A

4.1.1 The P_A Polyhedron

Consider the MIP model of the SDST flowshop given by (3.2.1)-(3.2.8). We are interested in the polyhedral description of the convex hull of the set of feasible solutions. Let $G_{n+1} = (V_{n+1}, A_{n+1})$ be a directed graph on $n+1$ nodes, where each node in the set V_{n+1} is associated with a job in J_0. We assume that G_{n+1} is complete. Thus $|A_{n+1}| = n(n+1)$. Let $X_{n+1} = \{x \in B^{n(n+1)} : x \text{ is the incidence vector of a tour in } G_{n+1}\}$.

Let

$$S_A = \{(x,y) \in B^{n(n+1)} \times R^{mn+1} : (x,y) \text{ is a feasible solution to (3.2.2)-(3.2.8)}\},$$

where the y vector includes the mn time variables (3.2.8) plus the makespan variable $C_{\max}$. Then S_A can be represented as follows:

$$S_A = \{(x,y) \, : \, x \in X_{n+1}, \, (x,y) \in C_A, \, y \in Y\},$$

where X_{n+1} is the set of constraints involving the binary variables only, $C_A = \{(x,y) : (x,y) \text{ satisfies (3.2.4)}\}$ is the set of coupling constraints involving both

binary and real variables, and $Y = \{y : y \text{ satisfies (3.2.5), (3.2.6), and (3.2.8)}\}$ is the set of constraints involving the real variables only. It is well known that the set X_{n+1} (the ATSP polytope on $n+1$ nodes) is characterized by (i) assignment constraints and (ii) subtour elimination constraints. In the formulation (3.2.2)-(3.2.8), the latter were omitted because they are implied by (3.2.4) which can be viewed as time-based subtour elimination constraints.

We are interested in the polyhedral structure of $P_A = \text{conv}(S_A)$, the convex hull of S_A. We have $n(n+1)$ binary variables (x_{jk}'s), and $mn+1$ nonnegative real variables (y_{ij}'s and $C_{\max}$) giving a total of $N = n(n+m+1)+1$ variables. Note that once a feasible incidence vector $x \in X_{n+1}$ has been determined, that is, once a given sequence is known, the computation of the associated $y \in R^{mn+1}$ that minimizes the makespan can be done recursively in $O(mn)$ operations.

The following proposition will be used for the main theorem which shows that P_A is full-dimensional.

Proposition 4.1 *Let θ be a positive real number, $y^0 \in R^t$ be a vector given by $y^0 = \theta\,(1, 2, \ldots, t \Leftrightarrow 1, t)^T$, and $y^u \in R^t$ be given by $y^u = y^0 + e^u$, where e^u is the u-th unit vector in R^t. Then, the vectors in the set $\{y^0, y^1, y^2, \ldots, y^t\}$ are affinely independent.*

Proof: For $\alpha_0, \alpha_1, \ldots, \alpha_p \in R$, we prove that the following system of linear equations

$$\sum_{u=0}^{t} \alpha_u y^u = 0 \tag{4.1}$$

$$\sum_{u=0}^{t} \alpha_u = 0 \tag{4.2}$$

implies $\alpha_u = 0$ for all $u = 0, \ldots, t$.

From (4.1) we have

$$\begin{aligned}
\sum_{u=0}^{t} \alpha_u y^u = 0 \quad &\Rightarrow \quad \alpha_0 y^0 + \sum_{u=1}^{t} \alpha_u (y^0 + e^u) = 0 \\
&\Rightarrow \quad \sum_{u=0}^{t} \alpha_u y^0 + \sum_{u=1}^{t} \alpha_u e^u = 0 \\
&\Rightarrow \quad y^0 \sum_{u=0}^{t} \alpha_u + \sum_{u=1}^{t} \alpha_u e^u = 0
\end{aligned}$$

From (4.2), we now have

$$\sum_{u=1}^{t} \alpha_u e^u = 0$$

so $\alpha_1 = \alpha_2 = \ldots = \alpha_n = 0$ and hence $\alpha_0 = 0$, which completes the proof. ■

We now state and prove the theorem defining the dimension of P_A.

Theorem 4.1 *Let $P_A = conv(S_A)$ be the convex hull of S_A. Then* $\dim(P_A) = n(n + m \Leftrightarrow 1)$

Proof: The proof consists of two parts.

(a) It is known that one of the $2(n + 1)$ assignment constraints (3.2.2)-(3.2.3) is redundant. This implies that $\text{rank}(A^=, b^=) \geq 2n + 1$, where $(A^=, b^=)$ is the equality set of P_A. It follows from Lemma B.1 that

$$\begin{aligned}
\dim(P_A) &\leq N \Leftrightarrow (2n + 1) \\
&= n(n + m + 1) + 1 \Leftrightarrow (2n + 1) \\
&= n(n + m \Leftrightarrow 1)
\end{aligned}$$

(b) To prove $\dim(P_A) \geq n(n + m \Leftrightarrow 1)$ we will show that there exists a set of $n(n + m \Leftrightarrow 1) + 1$ affinely independent vectors in R^N. In this regard, consider the subspace X_{n+1} of P_A. The dimension of the ATSP polyhedron on $n + 1$ vertices is $n^2 \Leftrightarrow n \Leftrightarrow 1$ (e.g., see [27]). This implies that there exists

a set of $K = n^2 \Leftrightarrow n$ affinely independent vectors $x^1, \ldots, x^K$ in $R^{n(n+1)}$, each being the incident vector of a tour. Also note that for any given $x^t \in X_{n+1}$, there exists a corresponding infinite number of feasible assignments of the time variables for P_A. For each x^t, $t = 2, \ldots, K$, let $y^t \in R^{mn+1}$ be any corresponding feasible assignment of the time variables on P_A. Here, y^t includes the mn time variables y_{ij}, and the makespan variable $C_{\max}$. Hence, the set S_1 given by

$$S_1 = \left\{ \begin{pmatrix} x^2 \\ y^2 \end{pmatrix}, \ldots, \begin{pmatrix} x^K \\ y^K \end{pmatrix} \right\}$$

is a set of feasible (and affinely independent) vectors in R^N, with $|S_1| = K \Leftrightarrow 1 = n^2 \Leftrightarrow n \Leftrightarrow 1$.

For x^1 we construct the corresponding y^1 as follows. Assume for simplicity that x^1 defines the job schedule $(1, 2, \ldots, n)$; that is, $x_{j,j+1} = 1$ for all $j = 0, 1, \ldots, n$ (indices 0 and $n+1$ are the same), and $x_{jk} = 0$ otherwise. Assume also that the $mn + 1$ components of y^1 are given in the order

$$y^1 = \begin{pmatrix} y^1_{11} \\ \vdots \\ y^1_{m1} \\ y^1_{12} \\ \vdots \\ y^1_{m2} \\ \vdots \\ y^1_{1n} \\ y^1_{mn} \\ C_{\max} \end{pmatrix}$$

That is, all the time variables associated with job 1 come first, then those for job 2, and so on, up to job n (the last in the sequence). The makespan variable comes at the end. Now, it is possible to select a large enough number θ such that the following yields a feasible solution for P_A:

$$\begin{pmatrix} y^1_{11} \\ y^1_{21} \\ \vdots \\ y^1_{m1} \\ \vdots \\ y^1_{1n} \\ \vdots \\ y^1_{mn} \\ C_{\max} \end{pmatrix} = \begin{pmatrix} \theta \\ 2\theta \\ \vdots \\ m\theta \\ \vdots \\ ((n \Leftrightarrow 1)m + 1)\theta \\ \vdots \\ mn\theta \\ (mn+1)\theta \end{pmatrix}$$

Let e^u be the u-th unit vector in R^{mn+1}, for all $u = 1, \ldots, mn+1$, and denote the vector $y^1 + e^u$ by $y^{1,u}$. By choosing θ as

$$\theta = \max_{ijk} \{p_{ij} + s_{ijk}\} + 1$$

we ensure not only the feasibility of y^1 but the feasibility of $y^{1,u}$ for all $u = 1, \ldots, mn+1$, as well. Using Proposition 4.1 with $t = mn + 1$ and y^1 as the base vector, we conclude that the $mn+2$ vectors in $\{y^1, y^{1,1}, y^{1,2}, \ldots, y^{1,mn+1}\}$ are affinely independent in R^{mn+1}, which in turn implies affine independence in R^N for the points in the set

$$S_2 = \left\{ \begin{pmatrix} x^1 \\ y^1 \end{pmatrix}, \begin{pmatrix} x^1 \\ y^{1,1} \end{pmatrix}, \begin{pmatrix} x^1 \\ y^{1,2} \end{pmatrix}, \ldots, \begin{pmatrix} x^1 \\ y^{1,mn+1} \end{pmatrix} \right\}$$

with $|S_2| = mn + 2$.

It is left to show that the vectors in $S_1 \cup S_2$ are affinely independent. Let α_t, β_u be real numbers for $t \in J_1 = \{1, \ldots, K\}$, and $u \in J_2 = \{1, \ldots, mn+1\}$ such that

$$\sum_{t \in J_1} \alpha_t \begin{pmatrix} x^t \\ y^t \end{pmatrix} + \sum_{u \in J_2} \beta_u \begin{pmatrix} x^1 \\ y^{1,u} \end{pmatrix} = 0 \qquad (4.3)$$

$$\sum_{t\in J_1} \alpha_t + \sum_{u\in J_2} \beta_u \quad = \quad 0 \tag{4.4}$$

This is a linear system of equations for (α_t, β_u). We now prove that this system has a unique zero solution. We distinguish three cases:

Case 1: $\alpha_t = 0$ for all $t \in J_1$

System (4.3)-(4.4) reduces to

$$\begin{aligned} \sum_{u\in J_2} \beta_u \begin{pmatrix} x^1 \\ y^{1,u} \end{pmatrix} &= 0 \\ \sum_{u\in J_2} \beta_u &= 0 \end{aligned}$$

Due to the affine independence of S_2, it follows that $\beta_u = 0$ for $u \in J_2$. Hence, an all-zero solution for (α_t, β_u) is obtained.

Case 2: $\beta_u = 0$ for all $u \in J_2$

The linear system (4.3)-(4.4) becomes

$$\begin{aligned} \sum_{t\in J_1} \alpha_t \begin{pmatrix} x^t \\ y^t \end{pmatrix} &= 0 \\ \sum_{t\in J_1} \alpha_t &= 0 \end{aligned}$$

which leads to $\alpha_t = 0$ for $t \in J_1$ due to the affine independence of the vectors in S_1.

Case 3: There exists $I_1, I_2 \neq \emptyset$ such that $\alpha_t \neq 0$ for all $t \in I_1 \subseteq J_1$ and $\beta_u \neq 0$ for all $u \in I_2 \subseteq J_2$. Here we have $\alpha_t = 0$ for all $t \in J_1 \setminus I_1$ and $\beta_u = 0$ for all $u \in J_2 \setminus I_2$. We show that Case 3 cannot occur. The corresponding linear system is

$$\sum_{t\in I_1} \alpha_t \begin{pmatrix} x^t \\ y^t \end{pmatrix} + \sum_{u\in I_2} \beta_u \begin{pmatrix} x^1 \\ y^{1,u} \end{pmatrix} \quad = \quad 0$$

$$\sum_{t\in I_1}\alpha_t+\sum_{u\in I_2}\beta_u \quad = \quad 0$$

which can be rewritten as

$$\sum_{t\in I_1}\alpha_t x^t + x^1\sum_{u\in I_2}\beta_u \quad = \quad 0 \tag{4.5}$$

$$\sum_{t\in I_1}\alpha_t y^t + \sum_{u\in I_2}\beta_u y^{1,u} \quad = \quad 0 \tag{4.6}$$

$$\sum_{t\in I_1}\alpha_t + \sum_{u\in I_2}\beta_u \quad = \quad 0 \tag{4.7}$$

We first note that $\beta' \equiv \sum_{u\in I_2}\beta_u \neq 0$. Otherwise (4.5) and (4.7) would become

$$\sum_{t\in I_1}\alpha_t x^t \quad = \quad 0$$

$$\sum_{t\in I_1}\alpha_t \quad = \quad 0$$

which implies, due to the affine independence of $\{x^t\}$, that $|I_1| = 0$. This is clearly a contradiction.

Now consider the following two subcases:

Case 3a: $1 \notin I_1$

Equations (4.5) and (4.7) become

$$\beta' x^1 + \sum_{t\in I_1}\alpha_t x^t \quad = \quad 0$$

$$\beta' + \sum_{t\in I_1}\alpha_t \quad = \quad 0$$

However, this contradicts the affine independence of $\{x^t\}$.

Case 3b: $1 \in I_1$

System (4.5)-(4.7) is rewritten as

$$(\alpha_1 + \beta')x^1 + \sum_{t \in I_1 \setminus \{1\}} \alpha_t x^t = 0 \qquad (4.8)$$

$$\sum_{t \in I_1} \alpha_t y^t + \sum_{u \in I_2} \beta_u y^{1,u} = 0 \qquad (4.9)$$

$$(\alpha_1 + \beta') + \sum_{t \in I_1 \setminus \{1\}} \alpha_t = 0 \qquad (4.10)$$

Equations (4.8) and (4.10), and the affine independence of $\{x^t\}$ imply that $I_1 \setminus \{1\} = \emptyset$; that is, $I_1 = \{1\}$ consists only of one index. Thus eqs. (4.9) and (4.10) become

$$\alpha_1 y^1 + \sum_{u \in I_2} \beta_u y^{1,u} = 0$$

$$\alpha_1 + \sum_{u \in I_2} \beta_u = 0$$

which contradicts the affine independence of $\{y^1, y^{1,u}\}$ (by Proposition 4.1).

This proves that Case 3 cannot occur.

The results from Cases 1 and 2 prove that $S_1 \cup S_2$ is an affine independent set in R^N, the size of set being $n(n + m \Leftrightarrow 1) + 1$. We conclude that $\dim(P_A) \geq n(n + m \Leftrightarrow 1)$.

Thus $\dim(P_A) = n(n + m \Leftrightarrow 1)$. ■

Corollary 4.1 *The equality set of P_A is given by the assignment constraints (3.2.2)-(3.2.3); that is,*

$$(A^=, b^=) = ((A^=_{\text{ATSP}}, 0), b^=)$$

where $A^=_{\text{ATSP}}$ is the equality set of the associated ATSP on $n + 1$ vertices.

Proof: Lemma B.1 and Theorem 4.1 imply that $\text{rank}(A^=, b^=) = 2n+1$, which is the rank of the equality set defined by the assignment constraints. ■

When a proper face F of P_A is found to have dimension $\dim(F) = n(n + m \Leftrightarrow 1) \Leftrightarrow 1$, Theorem 4.1 implies that F is a facet of P_A. We now establish the following relationship between facets of $\text{conv}(X_{n+1})$ (the ATSP polytope on $n+1$ nodes) and facets of P_A.

Theorem 4.2 *Let* $F_{\text{ATSP}} = \{x \in P \ : \ \pi x = \pi_0\}$ *be a facet of* $conv(X_{n+1})$. *Then*

$$F_A = \{(x, y) \in P_A \ : \ (\pi, 0)(x, y)^T = \pi_0\}$$

is a facet of P_A.

Proof: Let F_{ATSP} be a facet of $\text{conv}(X_{n+1})$. Then $\dim(F_{\text{ATSP}}) = \dim(T_{n+1}) \Leftrightarrow 1$, or, expressed in terms of the rank of its equality set,

$$\begin{aligned} \text{rank}\left(\begin{pmatrix} A^=_{\text{ATSP}} \\ \pi \end{pmatrix}, \begin{pmatrix} b^= \\ \pi_0 \end{pmatrix}\right) &= \text{rank}(A^=_{\text{ATSP}}, b^=) + 1 \\ &= 2n + 2 \end{aligned}$$

That is, (π, π_0) is linearly independent of the rows of $(A^=_{\text{ATSP}}, b^=)$. Note that $((\pi, 0), \pi_0)$ is a valid inequality for P_A and a nonempty face of P_A. Let $(A^=, b^=)$ be the equality set of P_A. Then $\text{rank}(A^=, b^=) = 2n + 1$. The equality set of F_A is then given by

$$(A^=_F, b^=_F) = \left(\begin{pmatrix} A^= \\ \pi' \end{pmatrix}, \begin{pmatrix} b^= \\ \pi_0 \end{pmatrix}\right)$$

where $\pi' = (\pi, 0)$. The rank of this equality set either stays the same at $(2n+1)$ or increases by one to $(2n + 2)$. Assume the former; i.e., that $\text{rank}(A^=_F, b^=_F) = 2n + 1$. This would imply that

$$\text{rank}\left(\begin{pmatrix} A^=_{\text{ATSP}} & 0 \\ \pi & 0 \end{pmatrix}, \begin{pmatrix} b^= \\ \pi_0 \end{pmatrix}\right) = 2n + 1$$

yielding

$$\text{rank}\left(\begin{pmatrix} A^{=}_{\text{ATSP}} \\ \pi \end{pmatrix}, \begin{pmatrix} b^{=} \\ \pi_0 \end{pmatrix}\right) = 2n+1;$$

which is a contradiction. Therefore, $\text{rank}(A^{=}_{F}, b^{=}_{F}) = 2n+2$, which gives $\dim(F_A) = \dim(P_A) \Leftrightarrow 1$; i.e., F_A is a facet of P_A. ■

This result is very important in the sense that any known facet of $\text{conv}(X_{n+1})$ can be easily transformed into a facet of P_A by just adding the corresponding zero vector ($0 \in R^{nm+1}$) to the inequality defining the facet in $R^{n(n+1)}$. The identification of such facets would be at the core of any B&C scheme devised to solve the SDST flowshop problem.

4.1.2 Lower Bound Mixed-Integer Cuts

For the purpose of developing cuts, we rewrite eqs. (3.2.4) and (3.2.8) as follows:

$$y_{ij} \Leftrightarrow y_{ik} + (A_i + \tau_{ijk})x_{jk} \leq A_i \quad i \in I,\ j,k \in J \tag{4.11}$$

$$B_{ij} \leq y_{ij} \quad i \in I,\ j \in J \tag{4.12}$$

where $\tau_{ijk} = p_{ij} + s_{ijk}$ accounts for both the processing and setup times on machine i. Let $z_{ij} = y_{ij} \Leftrightarrow B_{ij}$, so that $0 \leq z_{ij}$ and define $\xi_{ijk} = (A_i + \tau_{ijk})x_{jk}$. Substituting into (4.11) gives

$$z_{ij} \Leftrightarrow z_{ik} + \xi_{ijk} \leq A_i \Leftrightarrow B_{ij} + B_{ik} \tag{4.13}$$

Now, we apply Proposition B.1 with $N^+ = \{ij, ijk\}$, $N^- = \{ik\}$, $C = \{ij\}$, and $L = \emptyset$. If C is a dependent set; that is, if $\lambda = \tau_{ijk} + B_{ij} \Leftrightarrow B_{ik} > 0$, then (4.13) gives rise to the valid inequality

$$\xi_{ijk} + (A_i \Leftrightarrow B_{ij} + B_{ik})^+(1 \Leftrightarrow x_{jk}) \leq A_i \Leftrightarrow B_{ij} + B_{ik} + z_{ik} \tag{4.14}$$

Assuming $(A_i \Leftrightarrow B_{ij} + B_{ik})^+ > 0$, (4.14) becomes

$$\begin{aligned} \xi_{ijk} \Leftrightarrow (A_i \Leftrightarrow B_{ij} + B_{ik})x_{jk} &\leq z_{ik} \qquad \text{or} \\ (p_{ij} + s_{ijk} + B_{ij} \Leftrightarrow B_{ik})x_{jk} \Leftrightarrow y_{ik} &\leq \Leftrightarrow B_{ik} \end{aligned} \tag{4.15}$$

which is the desired result. Inequality (4.15) will have an effect only if $(p_{ij} + s_{ijk} + B_{ij} \Leftrightarrow B_{ik}) > 0$; that is, if C, as chosen, is a dependent set. Note that when $x_{jk} = 1$, (4.15) becomes $B_{ij} + p_{ij} + s_{ijk} \leq y_{ik}$ as expected and when $x_{jk} = 0$, it reduces to $B_{ik} \leq y_{ik}$, the default bound. We call this inequality a lower bound mixed-integer cut (LBMIC).

4.1.3 Upper Bound Mixed-Integer Cuts

Let z_{best} be the objective function value of the best known feasible schedule, i.e., a known upper bound on the optimal makespan. Then

$$C_{mj} \quad \leq \quad z_{best} \qquad j \in J$$

Let U_i denote an upper bound on the completion time of machine i and let

$$p^i_{min} \quad = \quad \min_j \{p_{ij}\} \qquad i \in I$$

By letting $U_m = z_{best}$, U_i can be recursively be computed as:

$$U_{i-1} \quad = \quad U_i \Leftrightarrow p^i_{min} \qquad i = m, m \Leftrightarrow 1, \ldots, 2$$

Then, inequality 3.2.4 can be strengthened by

$$y_{ij} + p_{ij} + s_{ijk} \quad \leq \quad y_{ik} + (U_i + s_{ijk})(1 \Leftrightarrow x_{jk})$$

Note that when $x_{jk} = 1$, the inequality will hold as expected, and when $x_{jk} = 0$ the inequality reduces to

$$y_{ij} + p_{ij} \quad \leq \quad y_{ik} + U_i$$

which will be redundant for all jobs (i, j, k) such that

$$C_{ij} = y_{ij} + p_{ij} \quad \leq \quad U_i$$

and will exclude schedules that have $C_{ij} > U_i$. This is fine, since this implies that schedule is suboptimal.

The U_i's are updated every time a new primal feasible solution is found. We called this inequalities the upper bound mixed-integer cut (UBMIC).

4.2 Polyhedral Results for Model B

Now consider the MIP model of the SDST flowshop given by (3.3.1)-(3.3.7). Let $S = \{S_i\}$, for $i = 1, 2, \ldots, n!$, be the set of all feasible schedules. For every schedule $S_i \in S$ there exists an incidence vector $x^i \in B^{n(n-1)/2}$. Let $\hat{X}_n = \{x \in B^{n(n-1)/2} : x \text{ is the incidence vector of a schedule}\}$.

Paralleling the notation in the previous section, let

$$S_B = \{(x, y) \in B^{n(n-1)/2} \times R^{mn+1} : (x, y) \text{ satisfies (3.3.2)-(3.3.7)}\}.$$

Again, the y vector includes the mn time variables (3.3.7) plus the makespan variable $C_{\max}$. The set S_B can be represented as follows: $S_B = \{(x, y) : x \in \hat{X}_n, (x, y) \in C_B, y \in Y\}$, where $\hat{X}_n$ is the set of constraints involving the binary variables only, $C_B = \{(x, y) : (x, y) \text{ satisfies (3.3.2)-(3.3.3)}\}$ is the set of coupling constraints involving both binary and real variables, and $Y = \{y : y \text{ satisfies (3.3.4), (3.3.5), and (3.3.7)}\}$ is the set of constraints involving the real variables only. Note that this set Y is the same as defined in the previous section.

We are interested in the polyhedral structure of $P_B = \text{conv}(S_B)$, the convex hull of S_B. Of particular interest is $\text{conv}(\hat{X}_n)$, the convex hull of $\hat{X}_n$ and its relationship to P_B. In contrast with formulation A, and the related polytope $\text{conv}(X_{n+1})$, the corresponding subspace $\hat{X}_n$ in formulation B has yet to be unexplored. In this section we first provide a more detailed study of the scheduling polyhedron $\text{conv}(\hat{X}_n)$. Subsequently, we give some results that link $\text{conv}(\hat{X}_n)$ with P_B, which in a sense, parallel those that allowed us to extend the polyhedral structure of $\text{conv}(X_{n+1})$ to P_A in the previous section.

4.2.1 The conv($\hat{X}_n$) Polyhedron

Throughout this section, we assume that the components of a feasible $x \in \hat{X}_n$ are stored columnwise; i.e., in the following order:

$$x = (x_{12}, x_{13}, x_{23}, \ldots, x_{1,n-1}, x_{2,n-1}, \ldots, x_{n-2,n-1}, x_{1,n}, x_{2,n}, \ldots, x_{n-1,n})^T$$

so $x \in B^{n(n-1)/2}$.

Lemma 4.1 *Conv($\hat{X}_n$) is full-dimensional; i.e.,* $\dim(\hat{X}_n) = \frac{n(n-1)}{2}$.

Proof: By induction on n. For $n = 2$ there are only two schedules, $S_1 = (1,2)$ and $S_2 = (2,1)$, with corresponding incidence one-dimensional vectors $x^1 = (1)$ and $x^2 = (0)$, respectively. Hence, $\text{conv}(\hat{X}_2)$ is given by $\text{conv}(\hat{X}_2) = \{x \in R : 0 \leq x \leq 1\}$. Clearly, $x = 1/2$ is an interior point of $\text{conv}(\hat{X}_2)$. It follows from Corollary B.1 that $\hat{X}_2$ is full-dimensional.

Now assume the induction hypothesis; that is, that $\text{conv}(\hat{X}_n)$ is full-dimensional. By implication there exists a set of $N+1$ affinely independent points $\{x^1, \ldots, x^N, x^{N+1}\}$, where $N = \dim(\hat{X}_n) = \frac{n(n-1)}{2}$ and each $x^i \in \hat{X}_n$ in the set is the incidence vector of a schedule. We need to prove that $\text{conv}(\hat{X}_{n+1})$ is full-dimensional.

In $\hat{X}_{n+1}$ there is an extra job to be scheduled (job $n+1$). The corresponding points have n additional coordinates with respect to the points in $\hat{X}_n$ given by the variables $x_{1,n+1}, x_{2,n+1}, \ldots, x_{n,n+1}$. Note that for any $x^i \in \hat{X}_n$, the assignment $x^i_{1,n+1} = x^i_{2,n+1} = \ldots = x^i_{n,n+1} = 0$ (which correspond to scheduling job $n+1$ at the beginning of S_i) yields a feasible schedule for $\hat{X}_{n+1}$ so

$$\left\{ \begin{pmatrix} x^1 \\ 0 \end{pmatrix}, \begin{pmatrix} x^2 \\ 0 \end{pmatrix}, \ldots, \begin{pmatrix} x^{N+1} \\ 0 \end{pmatrix} \right\} \subseteq \hat{X}_{n+1} \subseteq \text{conv}(\hat{X}_{n+1}).$$

Moreover, these vectors are affinely independent.

For a given $x^i \in X_n$, say x^1, we build n vectors in X_{n+1} as follows. Taking $x^1 \in X_n$ as a common base, we append the n-dimensional vector $v^j = (x^j_{1,n+1}, \ldots, x^j_{n,n+1})^T$, such that $(x^1, v^j)^T \in \hat{X}_{n+1}$, for $j = 1, \ldots, n$. Here, the components of v^j are determined when job $n+1$ is scheduled right after the

j-th scheduled job in S_1, for $j = 1, \ldots, n$. For instance, assuming for simplicity that x^1 is the incidence vector of $S_1 = (1, 2, \ldots, n)$, then

$$\begin{array}{lcll} \text{Insert } n+1 \text{ after } 1 & \Rightarrow & (1, n+1, 2, \ldots, n) & \Rightarrow \quad v^1 = (1, 0, \ldots, 0) \\ \text{Insert } n+1 \text{ after } 2 & \Rightarrow & (1, 2, n+1, 3, \ldots, n) & \Rightarrow \quad v^2 = (1, 1, 0, \ldots, 0) \\ \quad \vdots & & & \\ \text{Insert } n+1 \text{ after } n & \Rightarrow & (1, \ldots, n, n+1) & \Rightarrow \quad v^n = (1, 1, \ldots, 1). \end{array}$$

Note that the vectors in $\{v^j\}$ are linearly independent. The set

$$\left\{ \begin{pmatrix} x^1 \\ 0 \end{pmatrix}, \ldots, \begin{pmatrix} x^{N+1} \\ 0 \end{pmatrix}, \begin{pmatrix} x^1 \\ v^1 \end{pmatrix}, \ldots, \begin{pmatrix} x^1 \\ v^n \end{pmatrix} \right\}$$

has dimension $N+1+n = n(n \Leftrightarrow 1)/2+1+n = (n+1)n/2+1 = \dim(\hat{X}_{n+1})+1$. It remains to prove that these $N+1+n$ vectors are affinely independent. To do so, consider the following system of linear equations in (α_i, β_j), for $i = 1, \ldots, N+1$, $j = 1, \ldots, n$:

$$\begin{aligned} \sum_i \alpha_i \begin{pmatrix} x^i \\ 0 \end{pmatrix} + \sum_j \beta_j \begin{pmatrix} x^1 \\ v^j \end{pmatrix} &= 0 \\ \sum_i \alpha_i + \sum_j \beta_j &= 0 \end{aligned}$$

This system can be rewritten as

$$\sum_i \alpha_i x^i + \sum_j \beta_j x^1 = 0 \tag{4.16}$$

$$\sum_j \beta_j v^j = 0 \tag{4.17}$$

$$\sum_i \alpha_i + \sum_j \beta_j = 0. \tag{4.18}$$

Equation (4.17) and the fact that $\{v^j\}$ are linearly independent imply $\beta_j = 0$ for all j. Thus (4.16)-(4.18) reduces to

$$\sum_i \alpha_i x^i = 0$$

$$\sum_i \alpha_i \;\; = \;\; 0$$

It follows from the affine independence of $\{x^i\}$ that $\alpha_i = 0$ for all i. Therefore, the $(n+1)n/2+1$ vectors are affinely independent so $\dim(\text{conv}(\hat{X}_{n+1})) = (n+1)n/2$ implying that $\text{conv}(\hat{X}_{n+1})$ is full-dimensional. ■

4.2.2 Facets of conv($\hat{X}_n$)

We observe that $\hat{X}_n$ has certain symmetry in the sense that if $x \in \hat{X}_n$ then $\bar{x} \in \hat{X}_n$, where $\bar{x} = (1 \Leftrightarrow x_{12}, \ldots, 1 \Leftrightarrow x_{1n}, \ldots, 1 \Leftrightarrow x_{n-1,n})$ is the componentwise complement of x. This leads to the following lemma.

Lemma 4.2 *$F = \{x \in conv(\hat{X}_n) \;:\; \pi x = \pi_0\}$ is a facet of $conv(\hat{X}_n)$ if and only if $\bar{F} = \{x \in conv(\hat{X}_n) \;:\; \Leftrightarrow\pi x = \pi_0 \Leftrightarrow \sum_{jk} \pi_{jk}\}$ is a facet of $conv(\hat{X}_n)$, where $\sum_{jk} \pi_{jk}$ is the sum of all components of vector π.*

Proof: Since F is a facet of $\text{conv}(\hat{X}_n)$, $\dim(F) = \dim(\text{conv}(\hat{X}_n)) \Leftrightarrow 1$ (by Lemma B.1). Hence, there exists $K = \dim(\text{conv}(\hat{X}_n))$ affine independent vectors $x^i \in F$. Consider the vectors $\{\bar{x}^i\}$. It is easy to verify that $\bar{x}^i \in \bar{F}$. Furthermore, all the $\bar{x}^i$ are affinely independent as well, as shown below.

$$\begin{aligned}
\sum_i \alpha_i \bar{x}^i = 0 \quad \text{and} \quad \sum_i \alpha_i = 0 \;\; &\Rightarrow \;\; \sum_i \alpha_i (\mathbf{1} \Leftrightarrow x^i) = 0 \quad \text{and} \quad \sum_i \alpha_i = 0 \\
&\Rightarrow \;\; \mathbf{1} \sum_i \alpha_i \Leftrightarrow \sum_i \alpha_i x^i = 0 \quad \text{and} \quad \sum_i \alpha_i = 0 \\
&\Rightarrow \;\; \sum_i \alpha_i x^i = 0 \quad \text{and} \quad \sum_i \alpha_i = 0 \\
&\Rightarrow \;\; \alpha_i = 0 \quad \text{for all } i
\end{aligned}$$

due to the affine independence of the x^i vectors, where $\mathbf{1}$ is a vector with each component equal to 1. It follows that $\dim(\bar{F}) = K \Leftrightarrow 1$ and that $\bar{F}$ is a facet of $\text{conv}(\hat{X}_n)$. The converse is shown similarly. ■

Basically, Lemma 4.2 establishes that for every facet of conv($\hat{X}_n$) there is a symmetric counterpart which is also a facet of conv($\hat{X}_n$) and tells us how to find it.

Proposition 4.2 *The nonnegativity constraints*

$$x_{jk} \geq 0 \qquad j,k \in J, j < k$$

give facets of conv(X_n) for $n \geq 2$.

Proof: Let $j,k \in J, j < k$. Let $\pi x \leq 0$ represent the constraint $\Leftrightarrow x_{jk} \leq 0$, that is, $\pi = (0,\ldots,0,\Leftrightarrow 1,0,\ldots,0)$ and $\pi_0 = 0$ where the -1 component in π corresponds to π_{jk}. Note that

(a) $\pi x \leq \pi_0$ is a valid inequality of conv($\hat{X}_n$), so $F = \{x \in \text{conv}(\hat{X}_n) : \pi x = \pi_0\}$ is a face of conv(X_n).

(b) F is a proper face since $\pi x \leq \pi_0$ is satisfied at equality by some $x^i \in \hat{X}_n$ and is a strict inequality for some other $x^i \in \hat{X}_n$. In fact, any schedule S_i where job j is after (before) job k satisfies $\pi x \leq 0$ as an equality (strict inequality).

We prove the result by showing that conditions of Theorem B.1 hold. Here $\pi x \leq \pi_0$ represents a nonnegativity constraint, the equality set $(A^=, b^=)$ does not exist since conv(X_n) is full-dimensional, and we are concerned with solutions to the linear system

$$\lambda x^i = \lambda_0, \tag{4.19}$$

where x^i is the incidence vector of schedule S_i (with components stored row-wise) and $\{S_i\}$ is the set of schedules that satisfy $\pi x^i = \pi_0$. Hence, it suffices

to demonstrate that all solutions (λ, λ_0) to (4.19) for all i are of the form $\lambda = \alpha\pi,\ \lambda_0 = \alpha\pi_0$ for some $\alpha \in R$.

Because $\{S_i\}$ is the set of schedules satisfying $\pi x^i = \pi_0$, that is $x^i_{jk} = 0$, then $\{S_i\}$ contains all schedules where job k is scheduled before job j. In particular, $S_0 = (n, n \Leftrightarrow 1, \ldots, k, \ldots, j, \ldots, 2, 1) \in \{S_i\}$ and $x^0 = 0 \in B^{\frac{n(n-1)}{2}}$. Thus

$$\lambda x^0 = \lambda_0 \Leftrightarrow \lambda \cdot 0 = \lambda_0 \Leftrightarrow \lambda_0 = 0$$

so system (4.19) reduces to $\lambda x = 0$. To determine the solution

$$\lambda = (\lambda_{12}, \lambda_{13}, \ldots, \lambda_{1n}, \ldots, \lambda_{n-1,n}) \in R^{\frac{n(n-1)}{2}}$$

we proceed as follows. From S_0 we obtain S_1 by swapping jobs 2 and 1 such that

$$S_1 = (n, n \Leftrightarrow 1, \ldots, k, \ldots, j, \ldots, 3, 1, 2) \in \{S_i\}$$

with corresponding incidence vector $x^1 = (1, 0, \ldots, 0)$. Thus

$$\lambda x^1 = 0 \Leftrightarrow \lambda_{12} = 0.$$

Similarly, we obtain S_2 by swapping jobs 3 and 1:

$$S_2 = (n, n \Leftrightarrow 1, \ldots, k, \ldots, j, \ldots, 4, 1, 3, 2) \in \{S_i\}$$

with $x^2 = (1, 1, 0, \ldots, 0)$. Thus

$$\lambda x^1 = 0 \Leftrightarrow \lambda_{13} = 0$$

because we already have found that $\lambda_{12} = 0$.

Observe that every time we swap two adjacent jobs u, v, the corresponding incidence vectors are equal except for the component associated with these jobs x_{uv}. Also, as long as jobs j and k are not swapped, the resulting schedule remains feasible and satisfies $\pi x = \pi_0$. Therefore, by swapping job 1 with jobs $4, 5, \ldots, n$ (one at a time), we arrive at the schedule

$S_{n-1} = (1, n, n \Leftrightarrow 1, \ldots, k, \ldots, j, \ldots, 3, 2)$, finding along the way that $\lambda_{14} = \ldots = \lambda_{1n} = 0$; that is, $\lambda_{1q} = 0$ for all $q = 2, \ldots, n$.

Proceeding similarly with jobs $2, 3, \ldots, j \Leftrightarrow 1$, and evaluating (4.19) for each generated x^i, we find $\lambda_{pq} = 0$ for all $p = 1, \ldots, j \Leftrightarrow 1$ and $q = p+1, \ldots, n$. After the final swap, we have

$$S_l = (1, 2, \ldots, j \Leftrightarrow 2, j \Leftrightarrow 1, n, n \Leftrightarrow 1, \ldots, k, \ldots, j)$$

for some l.

By shifting one at a time the jobs in S_0 scheduled after job j, and by substituting the corresponding x^i in system (4.19), we have recursively found that $\lambda_{pq} = 0$ for all p, q such that $p < j$. If instead of shifting the jobs at the end of the schedule (after job j), we carry out the same procedure starting with the jobs at the beginning of the schedule (before job k) we arrive at the conclusion that $\lambda_{pq} = 0$ for all p, q such that $q > k$. That is, given S_l, swap jobs n and $n \Leftrightarrow 1$ to get

$$S_{l+1} = (1, \ldots, j \Leftrightarrow 1, n \Leftrightarrow 1, n, n \Leftrightarrow 2, \ldots, k, \ldots, j)$$

Then, $\lambda x^{l+1} = 0$ implies $\lambda_{n-1,n} = 0$. Keep on swapping job n with each of the jobs $n \Leftrightarrow 2, n \Leftrightarrow 3, \ldots, j$ one at a time to obtain $\lambda_{n-2,n} = \lambda_{n-3,n} = \ldots = \lambda_{j+1,n}, \lambda_{j,n} = 0$. After the last exchange, we have the schedule $S_{l+n-j} = (1, \ldots, j \Leftrightarrow 1, n \Leftrightarrow 1, n \Leftrightarrow 2, \ldots, k, \ldots, j, n)$. Repeat recursively this shifting procedure for jobs $n \Leftrightarrow 1, n \Leftrightarrow 2, \ldots, k+1$, to obtain $\lambda_{pq} = 0$ for all p, q such that $q = n, n \Leftrightarrow 1, \ldots, k+1$, with final schedule $S_r = (1, \ldots, j \Leftrightarrow 1, k, k \Leftrightarrow 1, \ldots, j+1, j, k+1, k+2, \ldots, n \Leftrightarrow 1, n)$, for some r.

It remains to determine λ_{pq} for all p, q such that $p = j, j+1, \ldots, k \Leftrightarrow 1$ and $q = p+1, \ldots, k$. However, by applying the same reasoning, we swap job k and $k \Leftrightarrow 1$ to get $\lambda_{k-1,k} = 0$. Then we swap job k with $k \Leftrightarrow 2$ and so on up to job $j+1$. This leads to $\lambda_{k-2,k} = \lambda_{k-3,k} = \ldots, \lambda_{j+1,k} = 0$ with the corresponding

schedule $S_{r+k-j} = (\ldots, k \Leftrightarrow 1, k \Leftrightarrow 2, k \Leftrightarrow 3, \ldots, j+1, k, j, k \Leftrightarrow 1, \ldots)$. Repeating these operations for job $k \Leftrightarrow 2, k \Leftrightarrow 3, \ldots, j+1$, but shifting all the way up to job j, we find $\lambda_{pq} = 0$ for all remaining (p,q) pairs except (j,k). The resulting schedule is $S_s = (1, \ldots, j \Leftrightarrow 1, k, j, j+1, \ldots, k \Leftrightarrow 1, k+1, \ldots, n)$, for some s. Therefore, $\lambda_{pq} = 0$ for all $(p,q) \neq (j,k)$.

Hence, a solution for (4.19) is given by $(\lambda, 0)$, where

$$\lambda = (0, \ldots, 0, \lambda_{jk}, 0, \ldots, 0).$$

It is straightforward to check that $\alpha = \Leftrightarrow \lambda_{jk}$ satisfies

$$\lambda = \alpha\pi \quad \text{and} \quad \lambda_0 = \alpha\pi_0$$

as was to be shown. ■

Corollary 4.2 *The inequalities*

$$x_{jk} \leq 1 \quad j,k \in J, j < k$$

give facets of conv(X_n) for all $n \geq 2$.

Proof: Follows from Proposition 4.2 and Lemma 4.2. ■

In contrast with X_{n+1} in model A, it is not possible to identify analogous ATSP valid inequalities such as subtour elimination constraints, comb inequalities, and D_k^+, D_k^- inequalities for model B. One set of valid inequalities that we can identify, though, corresponds to precedence violations for a sequence of jobs. Table 4.1 shows the valid inequalities that eliminate "cycles" (in the precedence sense) for any 3-job subsequence. We call these inequalities, for a subsequence of size t, the t-subsequence elimination constraint (or t-SEC). For $t = 3$ we show below that the 3-SEC are facets of conv($\hat{X}_n$).

Sequence	Constraint		
$j \to k \to l \Rightarrow j \to l$	$x_{jk} + x_{kl}$	$\leq$	$1 + x_{jl}$
$j \to l \to k \Rightarrow j \to k$	$x_{jl} + (1 - x_{kl})$	$\leq$	$1 + x_{jk}$

Table 4.1: 3-SECs for conv($\hat{X}_n$)

Lemma 4.3 *The inequalities (3-subsequence elimination constraints)*

$$x_{jk} \Leftrightarrow x_{jl} + x_{kl} \quad \geq \quad 0 \quad j,k,l \in J,\ j < k < l \tag{4.20}$$

give facets of conv($\hat{X}_n$) for all $n \geq 2$.

Proof: Each inequality in (4.20) represents a proper face of conv($\hat{X}_n$) since it is satisfied as an equality by some schedule (e.g., $S = (l,k,j,\ldots)$) and as a strict inequality for some other schedule (e.g., $S = (l,j,k,\ldots)$).

Again we prove the result by showing the conditions of Theorem B.1. Here, $\pi x \leq \pi_0$ is given by $\pi = (0,\ldots,0,\pi_{jk},0,\ldots,0,\pi_{jl},0,\ldots,0,\pi_{kl},0,\ldots,0)$ and $\pi_0 = 0$, where $\pi_{jk} = \pi_{jl} = \Leftrightarrow 1$ and $\pi_{kl} = 1$. Note that because conv($\hat{X}_n$) is full-dimensional, there is no equality set $(A^=, b^=)$.

Let $\{S_i\}$ be the set of schedules that satisfy $\pi x^i = \pi_0$, for all i. We are concerned with solutions to the linear system

$$\lambda x^i \quad = \quad \lambda_0 \tag{4.21}$$

where x^i is the incidence vector corresponding to schedule S_i. It suffices to demonstrate that all solutions (λ, λ_0) to (4.21) are of the form $\lambda = \alpha\pi, \lambda_0 = \alpha\pi_0$ for some $\alpha \in R$.

Equation $\pi x = \pi_0$ (that is, $x_{jk} \Leftrightarrow x_{jl} + x_{kl} = 0$) is satisfied if one of the following three cases occur:

(i) $x_{jk} = x_{jl} = x_{kl} = 0$, which corresponds to $S_i = (\ldots,l,\ldots,k,\ldots,j,\ldots)$.

(ii) $x_{jk} = 0, x_{jl} = x_{kl} = 1$ which corresponds to $S_i = (\ldots, k, \ldots, j, \ldots, l, \ldots)$.

(iii) $x_{jk} = x_{jl} = 1, x_{kl} = 0$, which corresponds to $S_i = (\ldots, j, \ldots, l, \ldots, k, \ldots)$.

Since $S_0 = (n, n \Leftrightarrow 1, \ldots, 2, 1) \in \{S_i\}$ (case (i)), then

$$\lambda x^0 = \lambda_0 \Leftrightarrow \lambda \cdot 0 = \lambda_0 \Leftrightarrow \lambda_0 = 0.$$

By performing the same job shifting procedure we used in the proof of Proposition 4.2 for the schedules associated with case (i), we find $\lambda_{pq} = 0$ for all $(p, q) \notin \{(j, k), (j, l), (k, l)\}$. Thus, (4.21) becomes

$$\lambda_{jk} x_{jk} + \lambda_{jl} x_{jl} + \lambda_{kl} x_{kl} = 0.$$

Case (ii) and (iii) imply

$$\begin{aligned} \lambda_{jl} + \lambda_{kl} &= 0 \\ \lambda_{jk} + \lambda_{jl} &= 0 \end{aligned}$$

which is a 2×3 system with solution $\lambda_{jl} = \beta, \lambda_{jk} = \lambda_{kl} = \Leftrightarrow\beta$ for any $\beta \in R$. Hence, by taking $\alpha = \beta$, (λ, λ_0) is given by $(\lambda, \lambda_0) = (\alpha\pi, \alpha\pi_0)$. This completes the proof. ■

Lemma 4.4 *The inequalities*

$$x_{jk} \Leftrightarrow x_{jl} + x_{kl} \leq 1 \quad j, k, l \in J,\ j < k < l$$

give facets of conv($\hat{X}_n$) for all $n \geq 2$.

Proof: Follows from Lemma 4.3 and Lemma 4.2. ■

All 4-SECs are shown in Table 4.2 for all $j, k, l, m \in J$, $j < k < l < m$. These valid inequalities, however, do not define facets of conv($\hat{X}_n$). In fact, because $\dim(\hat{X}_n) = n(n \Leftrightarrow 1)2$ and each 4-SEC can be expressed as the intersection of two of the previously developed facets of conv($\hat{X}_n$) (i.e., combinations of $x_{jk} \geq 0, x_{jk} \leq 1$, and 3-SEC), they define faces of dimension $n(n \Leftrightarrow 1)/2 \Leftrightarrow 2$.

Sequence	Constraint		
$j \rightarrow k \rightarrow l \rightarrow m \Rightarrow j \rightarrow m$	$x_{jk} + x_{kl} + x_{lm}$	$\leq$	$2 + x_{jm}$
$j \rightarrow k \rightarrow m \rightarrow l \Rightarrow j \rightarrow l$	$x_{jk} + x_{km} + (1 - x_{lm})$	$\leq$	$2 + x_{jl}$
$j \rightarrow l \rightarrow k \rightarrow m \Rightarrow j \rightarrow m$	$x_{jl} + (1 - x_{kl}) + x_{km}$	$\leq$	$2 + x_{jm}$
$j \rightarrow l \rightarrow m \rightarrow k \Rightarrow j \rightarrow k$	$x_{jl} + x_{lm} + (1 - x_{km})$	$\leq$	$2 + x_{jk}$
$j \rightarrow m \rightarrow k \rightarrow l \Rightarrow j \rightarrow l$	$x_{jm} + (1 - x_{km}) + x_{kl}$	$\leq$	$2 + x_{jl}$
$j \rightarrow m \rightarrow l \rightarrow k \Rightarrow j \rightarrow k$	$x_{jm} + (1 - x_{lm}) + (1 - x_{kl})$	$\leq$	$2 + x_{jk}$

Table 4.2: 4-SECs for conv($\hat{X}_n$)

The conv($\hat{X}_n$) polytope can be used to model other scheduling problems, such as single-machine and permutation flowshops problems, where every schedule is feasible. When real variables are introduced in the scheduling model, it remains to be determined whether the valid inequalities discussed above define facets of the complete polyhedron. In the next section we prove that this is the case for the SDST flowshop polyhedron.

4.2.3 The P_B Polyhedron

We now state and prove the theorem defining the dimension of P_B. The proof is very similar to the proof of Theorem 4.1 because a point $x \in \hat{X}_n$ defines a given feasible sequence for P_B just as $x \in X_{n+1}$ defines a feasible sequence for P_A; moreover, the definition of $y \in R^{mn+1}$ is the same for both polyhedrons.

Theorem 4.3 *Let $P_B = conv(S_B)$ be the convex hull of S_B. Then P_B is full-dimensional; i.e.,* $\dim(P_B) = n(n \Leftrightarrow 1)/2 + mn + 1$

Proof: Let $N = n(n \Leftrightarrow 1)/2 + mn + 1$. We will show that there exists a set of $N + 1$ affinely independent vectors in R^N.

Consider the subspace $\hat{X}_n$ of P_B. We proved in Lemma 4.1 that conv($\hat{X}_n$) is full-dimensional. This implies that there exists a set of $K = n(n \Leftrightarrow 1)/2 + 1$

affinely independent vectors $x^1, \ldots, x^K$ in $R^{n(n+1)}$, each being the incidence vector of a schedule. Also note that for any given $x^t \in \hat{X}_n$, there exists a corresponding infinite number of feasible assignments of the time variables for P_B.

From this point on the rest of the proof follows that of Theorem 4.1, part (b). We will just sketch the arguments. From the set $\{x^1, \ldots, x^K\}$ we build two disjoint sets $S_1, S_2 \subseteq R^N$ given by

$$S_1 = \left\{ \begin{pmatrix} x^2 \\ y^2 \end{pmatrix}, \ldots, \begin{pmatrix} x^K \\ y^K \end{pmatrix} \right\}$$

$$S_2 = \left\{ \begin{pmatrix} x^1 \\ y^1 \end{pmatrix}, \begin{pmatrix} x^1 \\ y^{1,1} \end{pmatrix}, \begin{pmatrix} x^1 \\ y^{1,2} \end{pmatrix}, \ldots, \begin{pmatrix} x^1 \\ y^{1,mn+1} \end{pmatrix} \right\}$$

where S_1 and S_2 are sets of feasible (and affinely independent) vectors in R^N, with $|S_1| = K \Leftrightarrow 1 = n(n \Leftrightarrow 1)/2$ and $|S_2| = mn + 2$, so that $|S_1 \cup S_2| = n(n \Leftrightarrow 1)/2 + mn + 2$. We then can prove that the points in $S_1 \cup S_2$ are affinely independent by showing that the linear system

$$\sum_{t \in J_1} \alpha_t \begin{pmatrix} x^t \\ y^t \end{pmatrix} + \sum_{u \in J_2} \beta_u \begin{pmatrix} x^1 \\ y^{1,u} \end{pmatrix} = 0$$

$$\sum_{t \in J_1} \alpha_t + \sum_{u \in J_2} \beta_u = 0$$

admits the unique solution $\alpha_t = \beta_u = 0$ for $t \in J_1 = \{1, \ldots, K\}$, and $u \in J_2 = \{1, \ldots, mn + 1\}$. This leads to conclude that $\dim(P_B) = n(n \Leftrightarrow 1)/2 + mn + 1$.

■

We now establish the following relationship between facets of $\text{conv}(\hat{X}_n)$ and facets of P_B.

Theorem 4.4 *Let $F_X = \{x \in conv(\hat{X}_n) : \pi x = \pi_0\}$ be a facet of $conv(\hat{X}_n)$. Then $F_B = \{(x, y) \in P_B : (\pi, 0)(x, y)^T = \pi_0\}$ is a facet of P_B.*

Proof: Let F_X be a facet of $\text{conv}(\hat{X}_n)$. Let (π', π_0) represent the inequality $\pi' z \leq \pi_0$ where $\pi' = (\pi, 0) \in R^N$ and $z = (x, y) \in P_B$. Hence F_B can be rewritten as $F_B = \{z \in P_B \ : \ \pi' z = \pi_0\}$. Given that F_X is a facet of $\text{conv}(\hat{X}_n)$, it follows that F_B is a proper face of P_B.

We prove the result by showing that conditions of Theorem B.1 hold. Here, the equality set $(A^=, b^=)$ does not exist since P_B is full-dimensional, and we are concerned with solutions to the linear system

$$\lambda z \quad = \quad \lambda_0 \tag{4.22}$$

where z is any point in P_B satisfying $\pi' z = \pi_0$. Hence, it suffices to demonstrate that all solutions (λ, λ_0) to (4.22) are of the form $\lambda = \alpha\pi, \lambda_0 = \alpha\pi_0$ for some $\alpha \in R$.

Since $z = (x, y) \in P_B$, the system in (4.22) can be rewritten as

$$\lambda_x x + \lambda_y y \quad = \quad \lambda_0. \tag{4.23}$$

Let $x^1 \in F_X$. According to the procedure described in the proof of Theorem 4.3, it is possible to construct $mn + 2$ feasible affinely independent points $y^0, y^1, \ldots, y^{mn+1}$, where $y^u = y^0 + e^u$ for all $u = 1, \ldots, mn + 1$. Here e^u denotes the u-th unit vector in R^{mn+1}. It easy to see that $z^i = (x^1, y^i) \in P_B$ for all $i = 0, \ldots, mn + 1$. Moreover, z^i satisfies $\pi' z^i = \pi x^1 = \pi_0$ for all i so that $z^i \in F_B$. Substituting these $mn + 2$ points in system (4.22) we have

$$\begin{aligned}
\lambda_x x^1 + \lambda_y y^0 &= \lambda_0 && (4.24)\\
\lambda_x x^1 + \lambda_y y^1 &= \lambda_0 && (4.25)\\
&\vdots \\
\lambda_x x^1 + \lambda_y y^{mn+1} &= \lambda_0 && (4.26)
\end{aligned}$$

By subtracting (4.24) from all other eqs. (4.25)-(4.26), we obtain the following system of order $mn + 1$:

$$\begin{aligned} \lambda_y(y^1 \Leftrightarrow y^0) &= 0 \\ &\vdots \\ \lambda_y(y^{mn+1} \Leftrightarrow y^0) &= 0 \end{aligned}$$

Since $y^i \Leftrightarrow y^0 = e^i$ it follows that $\lambda_y = 0 \in R^{mn+1}$. This reduces (4.22) to

$$\lambda_x x = \lambda_0$$

where x satisfies $\pi x = \pi_0$. Given that F_X is a facet, it follows that there is $\alpha \in R$ such that $\lambda_x = \alpha\pi, \lambda_0 = \alpha\pi_0$. This implies that $\lambda = (\lambda_x, \lambda_y) = (\alpha\pi, \alpha 0) = \alpha(\pi, 0) = \alpha\pi'$ and the proof is complete. ■

4.2.4 Lower Bound Mixed-Integer Cuts

Note that inequalities (3.3.2) and (3.3.7) in model B have the same structure as inequalities (3.2.4) and (3.2.8) in model A. Thus the valid inequality derived from these equations for model A also applies for model B; that is,

$$(p_{ij} + s_{ijk} + B_{ij} \Leftrightarrow B_{ik})x_{jk} \Leftrightarrow y_{ik} \leq \Leftrightarrow B_{ik} \tag{4.27}$$

is a valid inequality for model B. Recall that (4.27) will have an effect only if $(p_{ij} + s_{ijk} + B_{ij} \Leftrightarrow B_{ik}) > 0$. Note that when $x_{jk} = 1$, (4.27) becomes $B_{ij} + p_{ij} + s_{ijk} \leq y_{ik}$ as expected and when $x_{jk} = 0$, it reduces to $B_{ik} \leq y_{ik}$, the default bound.

In a similar fashion, we use inequalities (3.3.3) and (3.3.7), a change of variable $x'_{jk} = 1 \Leftrightarrow x_{jk}$ in (3.3.3), and the same procedure to derive the valid inequality

$$(p_{ik} + s_{ikj} + B_{ik} \Leftrightarrow B_{ij})(1 \Leftrightarrow x_{jk}) \Leftrightarrow y_{ij} \leq \Leftrightarrow B_{ij}$$

for model B, where again we must have $(p_{ik} + s_{ikj} + B_{ik} \Leftrightarrow B_{ij}) > 0$ for the inequality to be useful.

4.2.5 Upper Bound Mixed-Integer Cuts

Following the development of UB MICs for model A (Section 4.1.3), it is also possible to strengthen the representations of inequalities (3.3.2) and (3.3.3) when a given upper bound U_i for completion of machine i is known. In that section we showed how to compute U_i recursively from a given known upper bound in the value of the makespan. Then the following inequalities can be added to model B:

$$y_{ij} + p_{ij} + s_{ijk} \leq y_{ik} + (U_i + s_{ijk})(1 \Leftrightarrow x_{jk}) \quad (j,k) \in \hat{A}, i \in I \quad (4.28)$$

$$y_{ik} + p_{ik} + s_{ikj} \leq y_{ij} + (U_i + s_{ikj})x_{jk} \quad (j,k) \in \hat{A}, i \in I \quad (4.29)$$

When $x_{jk} = 1$, inequality (4.28) will hold and eq. (4.29) will reduce to

$$C_{ik} = y_{ik} + p_{ik} \leq y_{ij} + U_i$$

which will be redundant for all schedules with $C_{ik} \leq U_i$ and will exclude those schedules with $C_{ik} > U_i$. When $x_{jk} = 0$, the role of equations are reversed and we obtain identical results.

Chapter 5

Polyhedral Computations

5.1 Summary of Valid Inequalities

What distinguishes B&C from traditional cutting plane methods is that the inequalities generated are valid at each node of the search tree. In Chapter 4, we developed several valid inequalities for formulations A and B. We now summarize these results.

For model A, we showed that if $\{x \in P : \pi x = \pi_0\}$ is a facet of P, where P is the convex hull of the set of feasible solutions of a $(n+1)$-city ATSP, then

$$\{(x, y) \in P_A : (\pi, 0)(x, y)^T = \pi_0\}$$

is a facet of the convex hull of the set of feasible solutions of the SDST flowshop, where $x \in B^{n(n+1)}$ corresponds to an incidence vector of a tour in a $(n+1)$-city ATSP, $y \in R^{nm+1}$ is the vector of real-variables y in formulation A, and P_A denotes the convex hull of the set of feasible solutions of the SDST flowshop under formulation A. This result says that any of the facets developed for the ATSP can be applied to the SDST flowshop. In our work, we implemented subtour elimination constrains (SECs) and D_k^+ and D_k^- inequalities (e.g., see [27]) which are two of the most successful facets developed for the ATSP. Among these, we found that the SECs were much more effective. The D_k^+ and D_k^- inequalities had little or no impact on improving the polyhedral representation of the SDST flowshop polyhedron.

We also developed mixed-integer cuts (MICs) of the following form:

$$\begin{array}{lrcl} \text{(LBMICs)} & (p_{ij} + s_{ijk} + B_{ij} \Leftrightarrow B_{ik})x_{jk} \Leftrightarrow y_{ik} & \leq & \Leftrightarrow B_{ik} \qquad \text{and} \\ \text{(UBMICs)} & (U_i + s_{ijk})(1 \Leftrightarrow x_{jk}) + y_{ik} \Leftrightarrow y_{ij} & \geq & p_{ij} + s_{ijk} \end{array}$$

where, B_{ij} is a lower bound on y_{ij} as defined in Section 3.2, and U_i is an upper bound on the completion time of machine i.

For model B, we developed 3-subsequence elimination constraints (3-SECs), 4-subsequence elimination constraints (4-SECs), and both lower and upper bound mixed-integer inequalities. The k-SEC are inequalities that eliminate "cycles" (in the precedence sense) for any k-job subsequence. These are shown in Table 5.1, where B_{ij} and U_i are defined as before.

Cut type	Constraint		
3-SECs	$x_{jk} + x_{kl}$	$\leq$	$1 + x_{jl}$
	$x_{jl} + (1 - x_{kl})$	$\leq$	$1 + x_{jk}$
4-SECs	$x_{jk} + x_{kl} + x_{lm}$	$\leq$	$2 + x_{jm}$
	$x_{jk} + x_{km} + (1 - x_{lm})$	$\leq$	$2 + x_{jl}$
	$x_{jl} + (1 - x_{kl}) + x_{km}$	$\leq$	$2 + x_{jm}$
	$x_{jl} + x_{lm} + (1 - x_{km})$	$\leq$	$2 + x_{jk}$
	$x_{jm} + (1 - x_{km}) + x_{kl}$	$\leq$	$2 + x_{jl}$
	$x_{jm} + (1 - x_{lm}) + (1 - x_{kl})$	$\leq$	$2 + x_{jk}$
Lower bound MICs	$(p_{ij} + s_{ijk} + B_{ij} - B_{ik})x_{jk} - y_{ik}$	$\leq$	$-B_{ik}$
	$(p_{ik} + s_{ikj} + B_{ik} - B_{ij})(1 - x_{jk}) - y_{ij}$	$\leq$	$-B_{ij}$
Upper bound MICs	$(U_i + s_{ijk})(1 - x_{jk}) + y_{ik} - y_{ij}$	$\geq$	$p_{ij} + s_{ijk}$
	$(U_i + s_{ikj})x_{jk} + y_{ij} - y_{ik}$	$\geq$	$p_{ik} + s_{ikj}$

Table 5.1: Family of valid inequalities for model B

5.2 Separation Algorithms

For a given class of valid inequalities, the associated separation problem can be stated as follows: Given a point $\bar{x} \in R^p$ satisfying a certain subset of constraints, and a family F of SDST flowshop inequalities, find the most violated

member of F, i.e., an inequality $\alpha x \leq \alpha_0$ belonging to F and maximizing the degree of violation $\alpha\bar{x} \Leftrightarrow \alpha_0$. When this problem is solved optimally, we say that we have an exact separation algorithm. However, sometimes the separation problem is as difficult as the original problem so it is necessary to resort to heuristics to identify violated inequalities. Below we describe the procedures developed for models A and B.

5.2.1 Separation Procedures for SECs for Model A

Let $(\bar{x}, \bar{y}) \in R^{|A|} \times R^{mn+1}$ be a point satisfying constraints (3.2.2)-(3.2.6). This point is obtained by relaxing the integrality restriction on the binary variables x and solving the corresponding LP. As stated in Section 5.1, any facet for the ATSP is a facet for the SDST flowshop, where only the binary variables x are considered. Therefore, we drop the real variables y and are left with the problem of finding a violation of the classical TSP subtour elimination constraint

$$\sum_{(j,k)\in A\,:\,j,k\in W} \bar{x}_{jk} \quad \leq \quad |W| \Leftrightarrow 1 \tag{5.1}$$

for some $W \subseteq J, 2 \leq |W| \leq n \Leftrightarrow 1$, or prove that none exists. Note that (5.1) is equivalent to

$$\sum_{\substack{j\in W \\ k\in J\setminus W}} \bar{x}_{jk} + \sum_{\substack{j\in J\setminus W \\ k\in W}} \bar{x}_{jk} \quad \geq \quad 2 \tag{5.2}$$

SECs for the ATSP are symmetric inequalities, that is, inequalities of the form $\alpha x \leq \alpha_0$ with $\alpha_{jk} = \alpha_{kj}$ for all $(j,k) \in A$. Symmetric inequalities for the ATSP have a very important property. It has been shown [27] that there exists a correspondence between valid inequalities for the ATSP and valid inequalities for the symmetric TSP (STSP). If we define the mapping $f : R^A \rightarrow R^E$ (A is the arc set of the complete digraph and E is the edge set of the corresponding undirected graph) as follows: $f(\bar{x}) = \hat{x}$, where $\hat{x}_{jk} = \bar{x}_{jk} + \bar{x}_{kj}$ for all $j \neq k$, then

we have $f(P) = Q$, where P and Q are the polytopes of the ATSP and STSP, respectively. In other words, every inequality $\sum_{e \in E} \alpha_e \hat{x}_e \leq \alpha_0$ for STSP can be transformed into a valid ATSP inequality by simply replacing $\hat{x}_e$ by $x_{jk} + x_{kj}$ for all $e = (j,k) \in E$. This produces the symmetric inequality $\alpha x \leq \alpha_0$, where $\alpha_{jk} = \alpha_{kj}$ for all $j, k \in J, j \neq k$. Conversely, every symmetric ATSP inequality $\alpha x \leq \alpha_0$ corresponds to the valid STSP inequality $\sum_{e \in E} \alpha_e \hat{x}_e \leq \alpha_0$.

The above correspondence implies that every separation algorithm for STSP can be used, as a black box, for ATSP as well. Therefore, given the point $\bar{x}$, we first define the symmetric counterpart $\hat{x}$ of $\bar{x}$ by the transformation $\hat{x}_{jk} = \bar{x}_{jk} + \bar{x}_{kj}$ for all $j, k \in J$, and then apply a STSP separation algorithm to $\hat{x}$.

Now, let us define the undirected support graph of $\hat{x}$, denoted $G(\hat{x})$, as the graph formed by $n + 1$ vertices (n jobs plus a dummy job) and an edge (j,k) of weight $\hat{x}_{jk}$ for each $\hat{x}_{jk} > 0$. The problem of finding a violated SEC for STSP is equivalent to finding a cut in $G(\hat{x})$ that is less than 2. That is, given $\hat{x} \in R^E$ satisfying $0 \leq \hat{x}_{jk} \leq 1$ for all $(j,k) \in E$ and the assignment constraints (3.2.2)-(3.2.3), find a nonempty proper subset W of J such that

$$\sum_{\substack{j \in W \\ k \in J \setminus W}} \hat{x}_{jk} \quad < \quad 2 \tag{5.3}$$

holds, or prove that no such $W \subseteq J$ exists, where (5.3) is the violated version of (5.2) for the symmetric case.

Consequently, what we are interested in is finding a minimum capacity cut-set in the support graph $G(\hat{x})$ where the capacities are given by the weights $\hat{x}_{jk}$, $(j,k) \in E$. If the minimum cut-set in $G(\hat{x})$ has a capacity which is greater than or equal to 2, then we conclude that there exists no SEC that is violated by $\hat{x}$. Otherwise a vertex set W given by a minimum capacity cut-set defines a violated SEC.

To solve the separation problem, we use the MINCUT algorithm developed by Padberg and Rinaldi [55]. This algorithm has a time complexity of $O(n^4)$, which is the same complexity as the algorithm developed by Gomory and Hu [26]. However, empirical evidence over a large class of graphs has demonstrated the superiority of MINCUT over the Gomory-Hu procedure.

Example 5.1 Consider the following 7-job instance of $F2|s_{ijk},\ prmu|C_{\max}$.

p_{ij}	$j=1$	2	3	4	5	6	7
$i=1$	68	43	95	95	69	66	55
2	44	66	74	92	34	55	52

s_{1jk}	$k=1$	2	3	4	5	6	7
$j=0$	30	33	25	29	39	32	31
1	-	37	24	26	27	34	39
2	22	-	39	28	31	29	31
3	25	32	-	40	33	23	40
4	35	28	40	-	25	25	27
5	40	28	29	29	-	40	23
6	32	26	32	29	20	-	28
7	37	25	28	37	35	26	-

s_{2jk}	$k=1$	2	3	4	5	6	7
$j=0$	35	33	24	40	21	27	40
1	-	35	20	33	37	20	32
2	27	-	24	28	35	20	33
3	30	20	-	36	24	34	35
4	29	36	25	-	20	40	27
5	35	32	20	38	-	28	29
6	34	26	22	23	39	-	27
7	20	39	20	37	40	25	-

Suppose that at some node in the B&C search tree, the following fractional solution is obtained (LP relaxation):

$$\bar{x}_{12} = 0.8540 \quad \bar{x}_{40} = 1.0000 \quad \bar{y}_{11} = 30 \quad \bar{y}_{21} = 98$$

$$
\begin{array}{llllllll}
\bar{x}_{13} = 0.1460 & \bar{x}_{53} = 0.8540 & \bar{y}_{12} = 33 & \bar{y}_{22} = 76 \\
\bar{x}_{25} = 0.9113 & \bar{x}_{57} = 0.1460 & \bar{y}_{13} = 25 & \bar{y}_{23} = 120 \\
\bar{x}_{27} = 0.0887 & \bar{x}_{64} = 0.8723 & \bar{y}_{14} = 29 & \bar{y}_{24} = 124 \\
\bar{x}_{31} = 0.6375 & \bar{x}_{67} = 0.1278 & \bar{y}_{15} = 39 & \bar{y}_{25} = 108 \\
\bar{x}_{32} = 0.0386 & \bar{x}_{72} = 0.1074 & \bar{y}_{16} = 32 & \bar{y}_{26} = 98 \\
\bar{x}_{34} = 0.1278 & \bar{x}_{76} = 0.8926 & \bar{y}_{17} = 31 & \bar{y}_{27} = 164 \\
\bar{x}_{35} = 0.0887 & \bar{x}_{01} = 0.3625 & \bar{C}_{\max} = 216 & \\
\bar{x}_{36} = 0.1074 & \bar{x}_{07} = 0.6375 &
\end{array}
$$

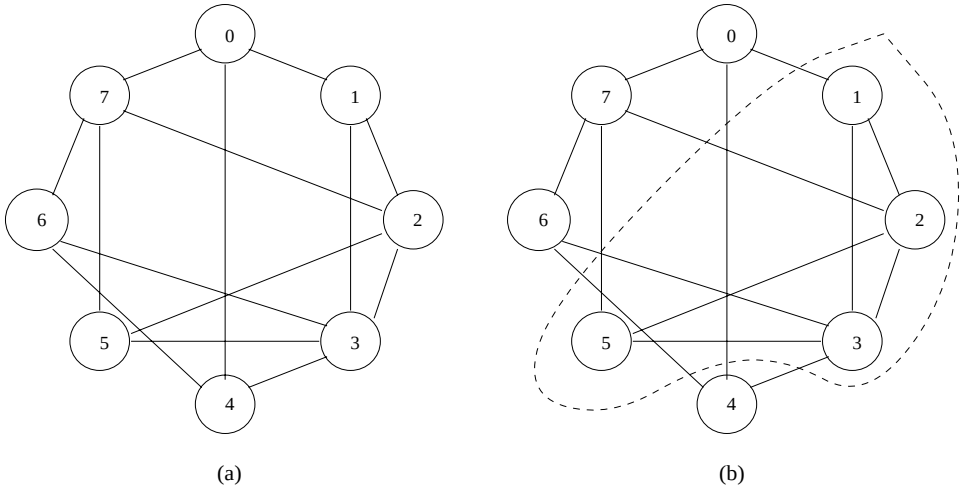


Figure 5.1: The support graph of $\hat{x}$

We transform $\bar{x}$ into its corresponding symmetric counterpart $\hat{x}$ using the transformation $\hat{x}_{jk} = \bar{x}_{jk} + \bar{x}_{kj}$ and then form $G(\hat{x})$, its support graph (depicted in Figure 5.1(a)). The edge weights are given by

Edge	Weight	Edge	Weight
(0,1)	0.3625	(2,7)	0.1961
(0,4)	1.0000	(3,4)	0.1278
(0,7)	0.6375	(3,5)	0.9427
(1,2)	0.8540	(3,6)	0.1074
(1,3)	0.7835	(4,6)	0.8723
(2,3)	0.0386	(5,7)	0.1460
(2,5)	0.9113	(6,7)	1.0204

By applying the MINCUT algorithm, we find that the minimum cut-set is given by $W = \{1, 2, 3, 5\}$ (shown in Figure 5.1(b)) with cut capacity equal to $x_{01} + x_{27} + x_{34} + x_{36} + x_{57} = 0.9398$. Since $0.9398 < 2$, the set W violates the following SEC:

$$\begin{aligned} & x_{12} + x_{13} + x_{15} + x_{21} + x_{23} + x_{25} \\ + & x_{31} + x_{32} + x_{35} + x_{51} + x_{52} + x_{53} \quad \leq \quad 3 = |W| \Leftrightarrow 1 \end{aligned}$$

for the ATSP. □

5.2.2 Separation Procedures for D_k^+ and D_k^- Inequalities

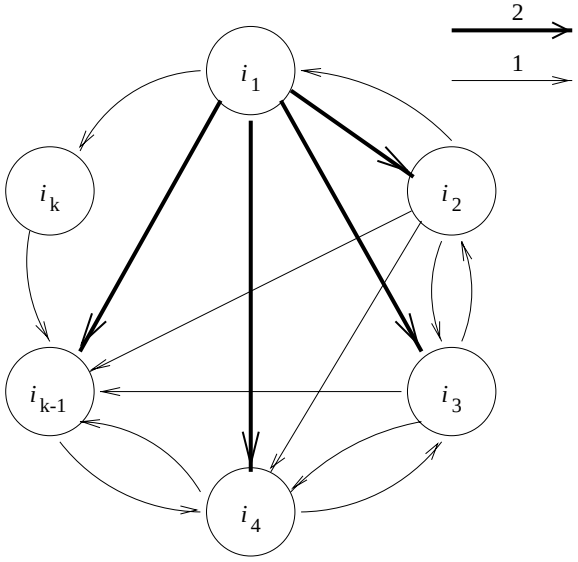


Figure 5.2: The Support Multigraph of a D_k^+ inequality

The following inequalities were derived by Grötschel and Padberg [27]:

$$(D_k^+) \quad x_{i_1 i_k} + \sum_{h=2}^{k} x_{i_h i_{h-1}} + 2\sum_{h=2}^{k-1} x_{i_1 i_h} + \sum_{h=3}^{k-1}\sum_{j=2}^{h-1} x_{i_j i_h} \quad \leq \quad k \Leftrightarrow 1 \qquad (5.4)$$

$$(D_k^-) \quad x_{i_k i_1} + \sum_{h=2}^{k} x_{i_{h-1} i_h} + 2\sum_{h=2}^{k-1} x_{i_h i_1} + \sum_{h=3}^{k-1}\sum_{j=2}^{h-1} x_{i_h i_j} \quad \leq \quad k \Leftrightarrow 1 \qquad (5.5)$$

where $(i_1 \ldots, i_k)$ is any sequence of $k \in \{3, \ldots, n \Leftrightarrow 1\}$ distinct nodes. A D_k^+ inequality for $k = 6$ is depicted in Figure 5.2, where arcs in dotted and solid line have coefficient 2 and 1, respectively. D_k^+ and D_k^- inequalities are facet-inducing for the ATSP polytope [24], and are obtained by lifting the cycle inequality $\sum_{(j,l) \in C} x_{jl} \leq k \Leftrightarrow 1$ associated with the circuit $C = \{(i_1, i_k), (i_k, i_{k-1}), \ldots, (i_2, i_1)\}$ and $C = \{(i_1, i_2), \ldots, (i_{k-1}, i_k), (i_k, i_1)\}$, respectively.

The separation problem for D_k^+ inequalities consists of finding a node sequence $(i_1, \ldots, i_k)$, $3 \leq k \leq n \Leftrightarrow 1$, such that (5.4) is violated. An exact enumeration scheme is proposed by Fischetti and Toth [25]. Here we use the following procedure. We first attempt to find all cycles in $G(\bar{x})$. Although the number of cycles in a complete graph may be large, usually $G(\bar{x})$ (coming from the SDST flowshop fractional solution) is relatively sparse, which allows us to identify the cycles in a relatively short amount of time. Then, for each cycle we attempt to find a violated D_k^+ and store the one with the largest degree of violation.

As pointed out by Fischetti and Toth [25], the D_k^- inequalities can be thought of as derived from D_k^+ inequalities by swapping the coefficient of the two arcs (j, k) and (k, j) for all $j, k \in J$, $j < k$. This is called a transposition operation. They show how this transposition enables the use of the separation procedures designed for D_k^+ inequalities as a separation procedure for D_k^- inequalities.

After implementing the D_k^+ and D_k^- separation procedures, we found they had very little impact on the overall performance of our B&C algorithm. Empirically, the SECs did a far better job in tightening the polyhedral representation. In our computations, only a very small number of D_k^+ and D_k^- inequalities were identified and, when added to the set of cuts, provided an insignificant improvement in the value of the LP relaxation.

Example 5.2 (Example 5.1 continued)
For the same fractional point $(\bar{x}, \bar{y})$, consider the following node sequence $(3, 5, 2, 1, 0, 4)$. Evaluating eq. (5.4) for $k = 6$ and $(i_1, \ldots, i_6) = (3, 5, 2, 1, 0, 4)$, we see that

$$\begin{aligned}
\bar{x}_{i_1 i_6} + \sum_{h=2}^{6} \bar{x}_{i_h i_{h-1}} + 2\sum_{h=2}^{5} x_{i_1 i_h} + \sum_{h=3}^{5}\sum_{j=2}^{h-1} x_{i_j i_h} &= (\bar{x}_{34} + \bar{x}_{40} + \bar{x}_{01} \\
&\quad + \bar{x}_{12} + \bar{x}_{25} + \bar{x}_{53}) \\
&\quad + 2(\bar{x}_{35} + \bar{x}_{32} + \bar{x}_{31} + \bar{x}_{30}) \\
&\quad + (\bar{x}_{52} + \bar{x}_{51} + \bar{x}_{21} \\
&\quad + \bar{x}_{50} + \bar{x}_{20} + \bar{x}_{10}) \\
&= (4.1096) + 2(0.7648) + (0.0) \\
&= 5.6392 > 5 = k \Leftrightarrow 1
\end{aligned}$$

is a violated D_6^+ inequality at $\bar{x}$. □

5.2.3 Separation Procedures for 3-SECs and 4-SECs for Model B

Given that there is a polynomial number ($O(n^3)$ and $O(n^4)$) of 3-SECs and 4-SECs (see Table 5.1), the corresponding separation problem can be solved optimally by simply looping over all indices for each type of 3-SECs (2 types) and 4-SECs (6 types). Empirically we found that the implementation of 4-SECs had very little or no impact at all on the performance of the B&C algorithm.

5.2.4 Separation Procedures for LBMICs and UBMICs

From Section 5.1 we can see that LBMICs for both models can be expressed in the following form:

$$\alpha_{ijk} x_{jk} \Leftrightarrow y_{ik} \quad \leq \quad \beta_{ijk}$$

where α_{ijk} and β_{ijk} are constants depending on problem data for $i \in I$ and $(j, k) \in A$ ($\hat{A}$) for model A (B). Thus given a point $(\bar{x}, \bar{y})$, by looping over all

possible index values i, j, k, we find the inequality such that

$$\alpha_{ijk}\bar{x}_{jk} \Leftrightarrow \bar{y}_{ik} \Leftrightarrow \beta_{ijk}$$

is maximized. This can be done in $O(mn^2)$ time.

Similarly, the UBMICs for both models can be expressed as

$$\gamma_{ijk}x_{jk} + y_{ik} \Leftrightarrow y_{ij} \quad \geq \quad \delta_{ijk}$$

where $\gamma_{ijk}, \delta_{ijk}$ are constants that depend on problem data. Again, by looping over all possible values of indices i, j, k the separation problem is solved exactly in $O(mn^2)$ time.

5.3 The Branch-and-Cut Method

Branch and cut (B&C) was introduced by Crowder and Padberg [14] who successfully solved large-scale instances of the well-known symmetric traveling salesman problem. It is considered state-of-the-art for the exact optimization of TSPs. The success of this method depends on the ability to find "strong" valid inequalities of the convex hull of the set of feasible solutions for a given mixed-integer program. This has been the case for the TSP, where many valid inequalities have been developed over the past 20 years. The SDST flowshop, however, has not been studied from a polyhedral perspective so one of our aims is to assess the effectiveness of B&C on this type of problem.

A typical B&B algorithm maintains a list of subproblems (nodes) whose union of feasible solutions contains all feasible solutions of the original problem. The list is initialized with the original problem itself. In each major iteration the algorithm selects a current subproblem from the list of unevaluated nodes. Typically in this subproblem, several of the binary variables have already been fixed to either zero or one when the node was generated. The algorithm solves the LP relaxation of this subproblem. This relaxation provides a lower bound

(for a minimization problem) for the original problem. Depending on the value of the solution, the node is either fathomed (e.g., if the relaxed LP is infeasible, or if the lower bound value exceeds the value of the best known feasible solution), which means that no further processing of the node is necessary, or split into new subproblems (children nodes) whose union of feasible solutions contains all feasible solutions of the current subproblem. These newly generated subproblems are added to the list of unevaluated subproblems.

Iterations are performed until the list of subproblems to be fathomed is empty. The crucial part of a successful B&B algorithm is the computation of the lower bounds. The better the LP-representation of the problem, the tighter the lower bound. This has a tremendous impact on the computational effort because it improves the chances that a node will be fathomed. Thus the corresponding portions of the search tree will not have to be evaluated. One way to improve the LP-representation of a given problem is by adding valid inequalities (cutting planes or cuts). B&C is the procedure developed to implement this idea.

Figure 5.3 shows a flow chart of our B&C algorithm which was coded within MINTO [52] using many of its built-in features. To discuss the relevant steps of the algorithm, the following notation is used: Z_{lp} is the objective function value of the current subproblem's LP relaxation, Z_{best} is the objective function value of the best feasible solution known so far, and Z_{heur} is the objective function value of a feasible solution delivered by a heuristic.

Read data: Read problem data and initialize the best global feasible solution value Z_{best} to infinity.

Preprocess: After the data have been read in, this stage attempts to improve the original formulation by removing redundant constraints and applying some probing techniques. The underlying idea of probing [64] is to analyze each of the inequalities of

the system of inequalities defining the feasible region in turn, trying to establish whether the inequality forces the feasible region to be empty, whether the inequality is redundant, whether the inequality can be used to improve the bounds on the variables, whether the inequality can be strengthened by modifying its coefficients, or whether the inequality forces some of the binary variables to either zero or one.

Select: A subproblem is chosen from the list of unevaluated candidates. Here we use a best-bound node selection strategy, which chooses the subproblem with the smallest lower bound.

Solve LP: The LP relaxation of the current subproblem is solved. We call its solution value Z_{lp}. If the problem is inconsistent or $Z_{lp} > Z_{best}$ the node is fathomed and we go back to the selection step. If the solution satisfies integrality and is feasible, then we update the current best global feasible solution (if $Z_{lp} < Z_{best}$), fathom the node, and go back to the selection step. Otherwise, we apply a heuristic in an attempt to find an integer feasible solution.

Primal heuristic: A heuristic is applied to see if it is possible to convert the current fractional solution to one that is integral. If successful, we update the current best global feasible solution (if $Z_{heur} < Z_{best}$), fathom the node, and go back to the selection step. In our implementation, we apply the SETUP heuristic (discussed in [62]) to the root node to start with a good feasible solution and use MINTO's built-in heuristic thereafter (invoked every 25 nodes).

Generate cuts: An attempt is made to identify a violated valid inequality. This is the most important component of the algorithm. The

generated inequalities are SECs (facet-inducing), D_k^+ and D_k^- inequalities (facet-inducing), LBMICs, and UBMICs for model A, and 3-SECs (facet-inducing), 4-SECs, UBMICs, and LBMICs for model B. If successful, we add the generated constraints to the formulation of the current subproblem and go back to solve the LP.

Branch: We need to specify how to partition (branch) the set of feasible solutions at the current node. For this type of formulation we do 0-1 variable fixing. This is based on fixing the value of a binary variable to either 0 or 1; i.e., two nodes are created. The way we determine the branching variable is by selecting the one with fractional value closest to $\frac{1}{2}$. The idea behind it is that it fixes a variable whose value in the optimal solution is hard to determine. The two newly created subproblems are added to the list of unevaluated nodes.

Although the conceptual algorithm stops when the list of unevaluated nodes is empty, we apply the following stopping criteria: (i) relative gap percentage; i.e., stop when a global integer feasible solution is within $\rho\%$ of optimality, (ii) time limit, and (iii) number of evaluated nodes limit.

5.4 Computational Evaluation

For the purpose of evaluating the B&C approach, we embedded all algorithmic components discussed above in MINTO (Mixed INTeger Optimizer [52]). MINTO is a shell that facilitates the development of implicit enumeration and column generation optimization algorithms that rely on linear relaxations. The user can enrich its basic features by providing a variety of specialized application functions to achieve maximum efficiency for a problem class. CPLEX [13]

was used to solve the LP relaxations. Our functions were written in C++ and linked to the MINTO 2.2 and CPLEX 4.0 libraries using the Sun compiler CC, version 2.0.1, with the optimization flag set to -O. CPU times were obtained through MINTO. The code was validated by solving several 100- and 150-job, 1-machine instances to optimality. Recall that the 1-machine problem is an ATSP.

To conduct our experiments we used randomly generated data from class D generator (see Appendix E),. It has been documented [30] that most real-world instances have a setup/processing time ratio between 20% and 40%. Class D tries to capture this behavior by randomly generating: $p_{ij} \in [20, 100]$ and $s_{ijk} \in [20, 40]$.

5.4.1 Experiment 1: B&B vs. B&C

In the first experiment our aim was to compare B&B with B&C. While it is true that B&C provides a stronger LP-representation, it also true that the size of the linear programs to be solved grows with the number of added cuts. Thus if the generated cuts are not especially effective, the resulting lower bound improvement will be more than offset by the corresponding increase in computational effort. To make this comparison, we generated 5 class D instances for each machine combination $m \in \{2, 4, 6\}$ and $n \in \{7, 8\}$, with a stopping limit of 90 CPU minutes. In a preliminary experiment we determined the most effective cuts for each model within the B&C framework. The best performance was observed using SECs and UBMICs for model A, and 3-SECs and the UBMICs for model B. The remaining computations were made with these cuts only.

Table 5.2 displays the results for models A and B for each machine instance. The problem size is given by number of constraints (NC), number of variables (NV), and number of nonzeros (NZ). The number of binary variables is given in parenthesis (B). The average algorithmic performance over the five

Instance size						Average performance			
$m \times n$	NC	NV(B)	NZ	Model	Method	Nodes	Cuts	LP rows	Time
2×7	114	71(56)	392	A	B&B	22687	0	114	10.1
				A	B&C	10091	129	236	6.7
	98	36(21)	280	B	B&B	11457	0	98	2.9
				B	B&C	7340	72	168	2.9
4×7	212	85(56)	672	A	B&B	21523	0	212	14.1
				A	B&C	9831	129	328	9.8
	196	50(21)	560	B	B&B	8392	0	196	3.6
				B	B&C	5261	73	266	3.4
6×7	310	99(56)	952	A	B&B	21635	0	310	20.2
				A	B&C	9864	132	435	14.1
	294	64(21)	840	B	B&B	9137	0	294	7.0
				B	B&C	5402	74	366	5.4

Table 5.2: Performance of B&B and B&C on 7-job class D instances for models A and B

instances is shown in terms of number of evaluated nodes (nodes), number of cuts added (cuts), maximum number of rows in the LP (LP rows), and CPU time in minutes. All instances were solved to optimality.

As can be seen, even though the size of the LPs increases (LP rows), the generated cuts are found to be effective on reducing the size of the feasible region as the B&C evaluates far fewer nodes and runs significantly faster. For model A the average relative time savings with B&C are 51%, 44%, and 43%, in the 2-, 4- and 6-machine instances, respectively. For model B, we observe little difference for the 2-, and 4-machine instances. The B&C starts to have an effect, however, as the size of the instance gets large. This can be seen in the 6-machine instances where B&C results in a relative time savings of 31%.

For model B, when we increase the number of jobs, B&C has a more pronounced impact. This can bee seen in Table 5.3 where the results for 8-job instances under model B are displayed. The B&C runs on average 33%,

Instance size					Average performance			
$m \times n$	NC	NV(B)	NZ	Method	Nodes	Cuts	LP rows	Time
2×8	128	45(28)	368	B&B	76096	0	128	61.4
				B&C	45072	114	138	46.0
4×8	256	61(28)	736	B&B	68579	0	256	68.6
				B&C	39149	116	366	55.3
6×8	384	77(28)	1104	B&B	59154	0	384	73.3
				B&C	34818	116	493	63.5

Table 5.3: Comparison of B&B and B&C on 8-job class D instances for model B

24%, and 15%, faster than the B&B on the 2-, 4-, and 6-machine instances, respectively. Table 5.4 displays the results when model A was used. As can be seen, the algorithm was unable to solve the problem (after 90 minutes) under either B&B or B&C. However, the optimality gaps (shown in the last column) are smaller under the latter. The relative optimality gap in MINTO is computed as follows:

$$\frac{\text{best upper bound} \Leftrightarrow \text{best lower bound}}{\text{best upper bound}} \times 100\%$$

Instance size					Average performance				
$m \times n$	NC	NV(B)	NZ	Method	Nodes	Cuts	LP rows	Time	Gap (%)
2×8	146	89(72)	512	B&B	50637	0	146	90.0	50.3
				B&C	49090	276	354	90.0	38.3
4×8	274	105(72)	880	B&B	45329	0	274	90.0	43.6
				B&C	39825	289	487	90.0	37.5
6×8	402	121(72)	1248	B&B	42719	0	402	90.0	39.6
				B&C	32486	287	607	90.0	36.3

Table 5.4: Comparison of B&B and B&C on 8-job class D instances for model A

5.4.2 Experiment 2: Model A vs. Model B

In Section 3.5 we pointed out the trade-off between models A and B. On one hand, model A can benefit from a better structured underlying TSP. In contrast, model B is smaller, using only about half the number of the binary variables used by model A.

By looking at the B&C rows for models A and B in Table 5.2 we can make a comparison of both models for 7-job instances. It can be seen that the size of the model (especially in terms of the number of binary variables) plays an important role. Computations are significantly better when model B is used. In fact, the effect is even more dramatic when we attempted to solve 8-job instances. By using model B, we were able to solve 8-job instances (Table 5.3) in an average of 46, 55.3, and 63.5 minutes of CPU for 2-, 4-, and 6-machines, respectively. When model A was used (Table 5.4), the algorithm stopped after 90 minutes with average optimality gaps of 38%, 37%, and 36%, respectively.

5.4.3 Experiment 3: Larger Instances

Instance size				Average performance			
$m \times n$	NC	NV(B)	NZ	Nodes	Cuts	LP rows	Gap (%)
2×10	200	66(45)	580	26428	241	412	34.8
4×10	400	86(45)	1160	20615	242	612	30.5
6×10	600	106(45)	1740	16453	241	812	26.7

Table 5.5: Evaluation of B&C on 10-job class D instances for model B

The last experiment assesses the limited scope of the polyhedral approach. Table 5.5 shows the average performance of B&C on 10-job instances with a 60-minute time limit for model B. We can see that the optimality gaps are 26-34%.

5.5 Conclusions

We provide empirical evidence that using model B with B&C yields better results on solving instances of the SDST flowshop problem for class D instances. The same results are observed when class A and C instances are used. However, the fact that even with the development of valid inequalities we are still unable to solve instances with 10 or more jobs shows that LP-based enumeration methods are wanting. The polyhedral representation of the problem is still not strong enough. In fact, we made several attempts to improve the performance of the B&C algorithm, such as changing branching strategies, fixing variables in a preprocessing phase, and reduced cost fixing, but the improvements were not significant. This difficulty is inherent to the SDST flowshop (2 or more machines) since we were able to successfully solve 100- and 150-job instances restricted to the 1-machine case. Recall that minimizing the makespan in SDST flowshop is equivalent to finding the minimum length tour of an $(n+1)$-city ATSP when the number of machines is set equal to 1. It is evident that once we start adding machines, the ATSP structure starts to weaken. One explanation for this is that, unlike the ATSP where we are looking for a good sequence of nodes, it is difficult here to characterize fully what a good sequence of jobs really is. What might be a good sequence for a certain machine, may be a bad sequence for the others. This makes this problem extremely nasty.

The quality of the LP relaxation lower bound led us to develop more efficient non-LP-based lower bounding procedures, which gave rise to a more effective enumeration scheme. This is the subject of Chapter 7.

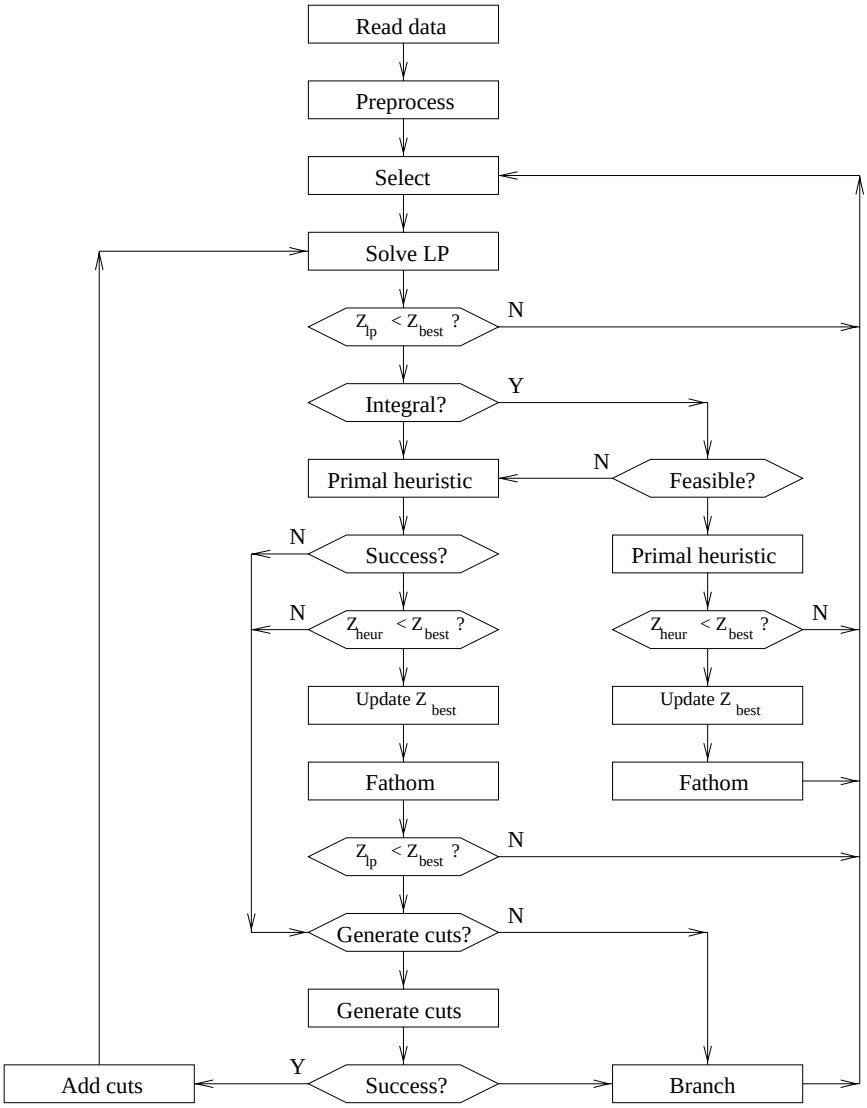


Figure 5.3: Flow chart of the B&C algorithm

Chapter 6

Heuristics

6.1 Preliminaries

In this chapter we present two heuristics for the SDST flowshop. `HYBRID()` is a deterministic heuristic that attempts to exploit the similarities between our problem and the ATSP. We also develop a randomized algorithm called `GRASP()`. Both heuristics are further enhanced by developing a family of local neighborhoods and implementing a corresponding local search procedure. A computational evaluation is given at the end of the chapter.

6.2 Hybrid Heuristic

6.2.1 ATSP-Based Heuristics

The best known heuristic for the SDST flowshop is due to Simons [65]. The main idea of his algorithm is first to transform an instance of the SDST flowshop into an instance of the ATSP by computing an appropriate cost matrix, and then to solve this by applying a well-known heuristic for the ATSP.

In the first of two phases of Simons' heuristics, an instance of the ATSP is built as follows. Every job is identified with a "city." Procedure TOTAL computes the entries in the distance (cost) matrix as the sum of both the processing and setup times over all the machines. Procedure SETUP considers the

sum of setup times only. In the second phase, a feasible tour is obtained by invoking a heuristic for the ATSP. This heuristic uses the well-known Vogel's approximation method (VAM) for obtaining good initial solutions to transportation problems with a slight modification to eliminate the possibility of subtours. The ATSP solution maps back into a feasible schedule for the SDST flowshop.

Although this approach seems suitable, given the strong similarities between the SDST flowshop and the ATSP, Simons' work was limited by the following two drawbacks. First, the cost function that penalizes scheduling two jobs together ignores completely the flowshop aspect of the problem; that is, there might be pairs of jobs that cause large amounts of blocking and/or machine idle time when they are scheduled together even though their setup times are small. In addition, no efforts were made to improve the solution by means of a local search procedure.

6.2.2 Description of Hybrid Heuristic

We attempt to improve Simons' idea by incorporating both the setup times and schedule fitness criteria in a penalty function between any pair of jobs. Let C_{jk} be the cost of scheduling job j right before job k. This measure can be expressed as

$$C_{jk} = \theta R_{jk} + (1 \Leftrightarrow \theta) S_{jk} \tag{6.1}$$

where $\theta \in [0, 1]$, and R_{jk} and S_{jk} are the costs of scheduling jobs j and k together, from the flowshop and the setup time perspective, respectively. The setup cost component is simply

$$S_{jk} = \sum_{i \in I} s_{ijk}$$

such that when $\theta = 0$, the cost measure is reduced to Simons' measure for his SETUP heuristic.

We now develop the cost R_{jk}. Here we follow an idea similar to the one used by Stinson and Smith [68] for $F||C_{\max}$. Let t_{ij} denote the completion of job j on machine i. Assume that job k immediately succeeds job j. The completion time of job k on any machine can then be recursively determined as follows:

$$t_{ik} \;=\; \max\{t_{ij} + s_{ijk}, t_{i-1,k}\} + p_{ik}$$

The relationship between $t_{ij}+s_{ijk}$ and $t_{i-1,k}$ plays a key role here. If $t_{ij}+s_{ijk} > t_{i-1,k}$, then job k will arrive at machine i before job j has released machine i; hence job k will be *blocked* in the queue at machine i for $t_{ij} + s_{ijk} \Leftrightarrow t_{i-1,k}$ time units. On the other hand, if $t_{ij} + s_{ijk} < t_{i-1,k}$, then machine i will be idle for $t_{i-1,k} \Leftrightarrow (t_{ij} + s_{ijk})$ time units while waiting for job k to arrive. The ideal situation, of course, will occur when $t_{ij} + s_{ijk} = t_{i-1,k}$ where neither a block to job k nor idleness to machine i would result.

Now, let us take this rationale a step further by considering the set of circumstances which would have to take place if $t_{ij} + s_{ijk}$ were to ideally equal $t_{i-1,k}$ for the entire period where both j and k are jointly in process in the schedule. Clearly, this will occur when $p_{ij} + s_{ijk} = s_{i-1,jk} + p_{i-1,k}$ for every machine $i = 2, \ldots, m$. Although we would seldom, if ever, expect such an ideal set of circumstances in practice, we still may recognize that the closer we can match the sets of $p_{ij} + s_{ijk}$ and $s_{i-1,jk} + p_{i-1,k}$ values for all machines, the smoother jobs j and k will tend to fit together within the schedule. We now define a residual, r_{ijk} as

$$r_{ijk} \;=\; p_{ij} + s_{ijk} \Leftrightarrow (s_{i-1,jk} + p_{i-1,k}) \qquad i \in I \setminus \{1\},\; j,k \in J$$

For any pair (j,k), $j \neq k$, we may compute $m \Leftrightarrow 1$ such residuals. These residuals are then heuristically combined to yield the overall cost, R_{jk}. The following choices were considered

Rule 1: Sum of the absolute residuals (R1)

$$R^1_{jk} = \sum_{i=2}^{m} |r_{ijk}|$$

Rule 2: Sum of positive residuals only (R2)

$$R^2_{jk} = \sum_{i=2}^{m} [r_{ijk}]^+$$

where $[r_{ijk}]^+ = r_{ijk}$ if $r_{ijk} > 0$, 0 otherwise.

Rule 3: Sum of negative residuals only (R3)

$$R^3_{jk} = \sum_{i=2}^{m} [r_{ijk}]^-$$

where $[r_{ijk}]^- = \Leftrightarrow r_{ijk}$ if $r_{ijk} < 0$, 0 otherwise.

Rule 4: Sum of absolute residuals with positive residuals weighted double (R4)

$$R^4_{jk} = \sum_{i=2}^{m} 2[r_{ijk}]^+ + [r_{ijk}]^-$$

Rule 5: Sum of absolute residuals with negative residuals weighted double (R5)

$$R^5_{jk} = \sum_{i=2}^{m} [r_{ijk}]^+ + 2[r_{ijk}]^-$$

With R1 each residual, regardless of its direction of error, is equally weighed. Rules R2 and R4 penalize more for positive residuals (blocking) whereas R3 and R5 penalize more for negative residuals (idle time). It is important to note that the sign of each r_{ijk} value is significant. A positive r_{ijk} implies that a degree of blocking for job k at machine i is likely to occur. On the other hand, a negative r_{ijk} implies idleness at machine i. This motivates the choices for rules R2-R5. Preliminary computational experience has shown that

rules R2 and R4 (which penalize more for positive residuals) are totally dominated by the other rules. This indicates that it is more serious to incur machine idleness than job blocking. One explanation for this is that a negative residual at some machine i, has a carryover effect on other machines downstream of i.

As far as the weight θ in eq. (6.1) is concerned, preliminary computational testing has shown that the best schedules are found when $\theta \in [0, 0.2]$. Note that for a given value of $\theta \in [0, 0.2]$ and residual cost rule, there is an associated cost matrix C. This suggests the following hybrid heuristic.

Procedure `HYBRID_phase1()`

Input: An instance of the SDST flowshop, a discretization Θ of the weight range, and a set R of residual cost functions.

Output: A feasible schedule S.

0: Initialize best schedule $S_{best} = \emptyset$
1: for each $\theta \in \Theta$ do
2: for each $R^i \in R$ do
3: Compute $(n+1) \times (n+1)$ cost matrix as $C_{jk} = \theta R^i_{jk} + (1 \Leftrightarrow \theta) S_{jk}$
4: Apply VAM to (C_{jk}) to obtain a tour S
5: If $C_{\max}(S) < C_{\max}(S_{best})$ then $S_{best} \leftarrow S$
6: Output S_{best}
7: Stop

Figure 6.1: Pseudocode of `HYBRID()` phase 1

Let $\Theta = \{\theta_1, \ldots, \theta_p\}$ be a (finite) discretization of $[0, 0.2]$, where p is the size of the discretization, and let $R = \{R^1, R^3, R^5\}$ be the set of cost functions (as defined above). The construction phase of procedure `HYBRID()` is shown in

Figure 6.1. A local search phase is then applied to this schedule to attempt to find a local optimum with respect to a determined neighborhood. Local search procedures are discussed in Section 6.4.

Computational complexity: The computation of the cost matrix performed in Step 3 takes $O(mn^2)$ time. The application of Voguel's method to a $(n+1)$-city problem is $O(n^2)$ and hence the overall procedure have worst-case complexity of $O(|R||\Theta|mn^2)$. Since $|R| = O(1)$ this brings the complexity down to $O(|\Theta|mn^2)$. Now, preliminary computational experience has convincingly shown that any discretization with $|\Theta| > 3$ provides no better solutions than a discretization with $|\Theta| = 3$. Hence, we take $\Theta = \{0.0, 0.1, 0.2\}$ and this procedure has a time complexity of $O(mn^2)$.

6.3 GRASP

6.3.1 General Methodology

A greedy randomized adaptive search procedure (GRASP), is a heuristic approach to combinatorial optimization problems that combines greedy heuristics, randomization and local search techniques. GRASP has been applied successfully to set covering problems that arise from the incidence matrix of Steiner triple systems (Feo and Resende [21]), airline flight scheduling and maintenance base planning (Feo and Bard [18]), scheduling on parallel machines (Laguna and González-Velarde [43]), railroad hitch assignment (Feo and González-Velarde [20]), p-hub location problems (Klincewicz [40]), single machine scheduling (Feo et al. [19]), maximum independent set problems (Feo et al. [23]), quadratic assignment problems (Li et al. [45] and Mavridou et al. [48]), graph planarization (Resende and Ribeiro [61]), and vehicle routing problems with time windows (Kontoravdis and Bard [41]).

GRASP consists of two phases: a construction phase and a postpro-

cessing phase. During the construction phase, a feasible solution is built, one element (job) at a time. At each iteration, all feasible moves are ranked and one is randomly selected from a restricted candidate list (RCL). The ranking is done according to a greedy function that adaptively takes into account changes in the current state.

One way to limit the RCL is by its cardinality where only the top λ elements are included. A different approach is by considering only those elements whose greedy function value is within a fixed percentage of the best move. Sometimes both approaches are applied simultaneously; i.e., only the top λ elements whose greedy function value is within a given percentage ρ of the value of the best move are considered. The choice of the parameters λ and ρ requires insight into the problem. A compromise has to be made between being too restrictive or being too inclusive. If the criterion used to form the list is too restrictive, only a few candidates will be available. The extreme case is when only one element is allowed. This corresponds to a pure greedy approach so the same solution will be obtained every time GRASP is executed. The advantage of being restrictive in forming the candidate list is that the greedy objective is not overly compromised; the disadvantage is that the optimum and many very good solutions may be overlooked.

GRASP phase 1 is applied N times, using different initial seed values to generate a solution (schedule) to the problem. In general, a solution delivered in phase 1 is not guaranteed to be locally optimal with respect to simple neighborhood definitions. Hence it is often beneficial to apply a postprocessing phase (phase 2) where a local search technique is used to improve the current solution. In our implementation, we apply the local search every $K = 10$ iterations to the best phase 1 solution in that subset. The procedure outputs the best of the N/K local optimal solutions. Figure 6.2 shows a flow chart of our implementation.

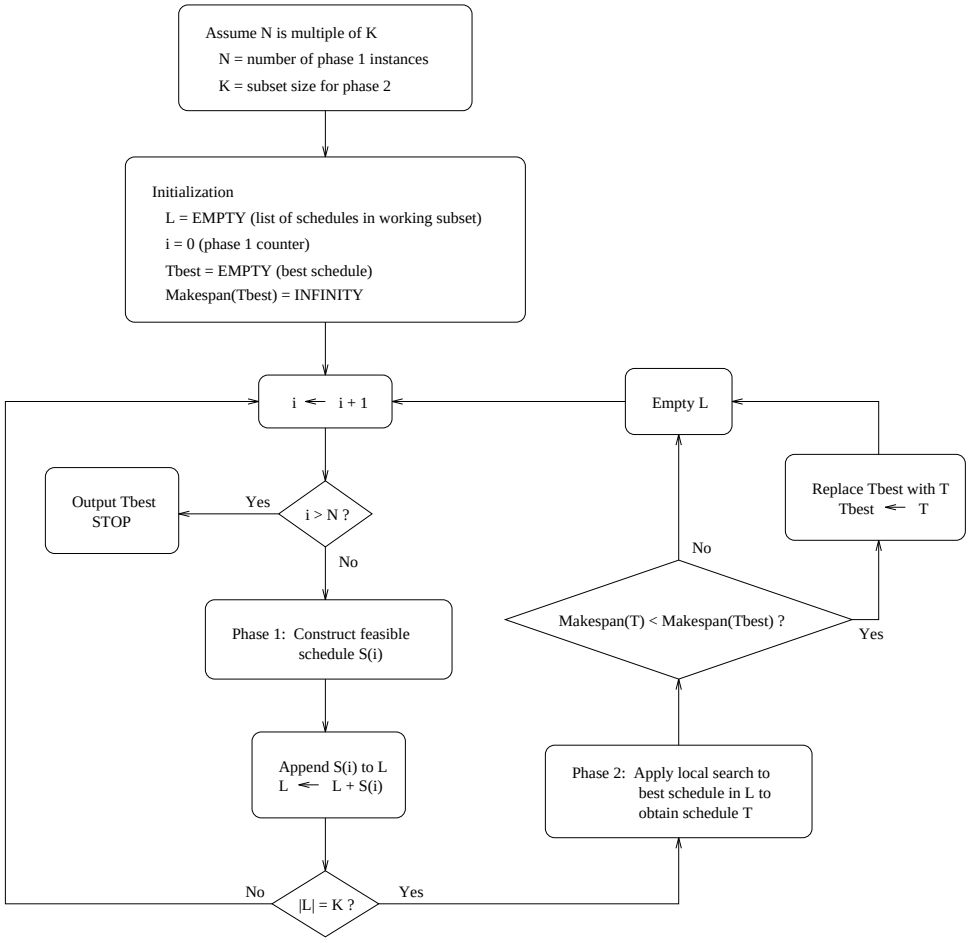


Figure 6.2: Flow chart of complete GRASP algorithm

The fundamental difference between GRASP and other meta-heuristics such as tabu search and simulated annealing is that GRASP relies on high quality phase 1 solutions (due to the inherent worst-case complexity of the local search) whereas the other methods do not depend on good feasible solutions. They spend practically all of their time improving the incumbent solution and attempting to overcome local optimality. For a GRASP tutorial, the reader is referred to [22].

6.3.2 GRASP for the SDST Flowshop

The best known heuristic for the general flowshop scheduling problem with makespan minimization is due to Nawaz et al. [51]. This procedure consists of inserting a job into the best available position of a set of partially scheduled jobs; that is, in the position that would cause the smallest increment on the value of the makespan. The original worst-case complexity of the heuristic was $O(mn^3)$. Later, Taillard [70] proposed a better way to perform the computations and came up with a complexity of $O(mn^2)$. Here, we use Taillard's idea extending it to handle setup times appropriately within the GRASP frame.

`GRASP()` construction phase is described as follows. At each iteration of the algorithm there is a partial schedule S. A job h is selected from a priority list P of unscheduled jobs. Nawaz et al. suggest an LPT (largest processing time) priority list; that is, a list where the jobs are ordered from largest to smallest total processing time. The partial schedule S and the job h define a unique greedy function $\psi(j) : \{0, 1, \ldots, |S|\} \rightarrow R$, where $\psi(j)$ is the makespan of the new schedule S' resulting from inserting job h at the j-th position (right after the j-th job) in S. Here, position 0 means an insertion at the beginning of the schedule.

In `GRASP()`, the positions available for insertion are sorted by nondecreasing values of $\psi(j)$ and a restricted candidate list is formed with the best λ positions. Preliminary testing has shown that for this type of scheduling problem, $\lambda = 2$ works best. The probabilistic strategy of `GRASP()` selects one of the positions in the RCL randomly with equal probability. The job h is inserted at the selected position into the current partial schedule S and the completion times C_{ij} for all jobs in the schedule are updated. Figure 6.3 shows the pseudocode of the procedure (phase 1).

In Step 1 of `GRASP()`, we form an LPT (largest processing time) priority list with respect to the sum of the processing times of each job over all the

Procedure `GRASP_phase1()`

Input: An instance of the SDST flowshop, a set P of unscheduled jobs, and size λ of the restricted candidate list.

Output: A feasible schedule S.

0: Set $S = \emptyset$
1: Sort the jobs in P to form an LPT priority list
2: while $|P| > 0$ do
2a: Remove h, the first job from P
2b: Compute $\psi(j)$ for position $j = 1, \ldots, |S + 1|$
2c: Construct the RCL with the best λ positions
2d: Choose randomly a position k from RCL
2e: Insert job h at position k in S
3: Output S
4: Stop

Figure 6.3: Pseudocode of `GRASP()` phase 1

machines. In Step 2b, we use a modification of Taillard's [70] procedure. Our modification, which is described next, includes sequence-dependent setup times.

Computing the partial makespans: We now describe how to efficiently compute the greedy function $\psi(j)$ given in Step 2b of `GRASP()` (Figure 6.3). Typically, a job within brackets $[j]$ denotes the job in position j. Here, for simplicity, we drop the brackets and assume that a current schedule is given by $S = (1, 2, \ldots, k \Leftrightarrow 1)$. Let h denote the job to be inserted. Define the following parameters:

- e_{ij} = the earliest completion time of the j-th job on the i-th machine;

Procedure `Makespans()`

Input: A partial schedule $S = (1, 2, \ldots, k \Leftrightarrow 1)$ and job k to be inserted.

Output: A vector $\psi(j)$ with the value of the makespan when job k is inserted in the j-th position of schedule S.

1: Compute the earliest completion times e_{ij}
2: Compute the tails q_{ij}
3: Compute the relative completion times f_{ij}
4: Compute values of partial makespan $\psi(j)$
5: Output vector $\psi(j)$
6: Stop

Figure 6.4: Pseudocode of procedure for computing partial makespans

$(i = 1, 2, \ldots, m)$ and $(j = 1, 2, \ldots, k \Leftrightarrow 1)$. These parameters are recursively computed as

$$\begin{aligned} e_{i0} &= 0 \\ e_{0j} &= r_j \\ e_{ij} &= \max\{e_{i-1,j},\ e_{i,j-1} + s_{i,j-1,j}\} + p_{ij} \end{aligned}$$

where r_j denotes the release time of job j. Here r_j is assumed to be zero.

- q_{ij} = the duration between the starting time of the j-th job on the i-th machine and the end of operations; $(i = m, m \Leftrightarrow 1, \ldots, 1)$ and $(j = k \Leftrightarrow 1, k \Leftrightarrow 2, \ldots, 1)$.

$$\begin{aligned} q_{ik} &= 0 \\ q_{m+1,j} &= 0 \\ q_{ij} &= \max\{q_{i+1,j},\ q_{i,j+1} + s_{i,j,j+1}\} + p_{ij} \end{aligned}$$

- f_{ij} = the earliest relative completion time on the i-th machine of job h inserted at the j-th position; $(i = 1, 2, \ldots, m)$ and $(j = 1, 2, \ldots, k)$.

$$\begin{aligned} f_{i0} &= 0 \\ f_{0j} &= r_h \\ f_{ij} &= \max\{f_{i-1,j},\ e_{i,j-1} + s_{i,j-1,h}\} + p_{ih} \end{aligned}$$

- $\psi(j)$ = the value of the partial makespan when adding job h at the j-th position; $(j = 1, 2, \ldots, k)$.

$$\psi(j) = \max_{i=1,\ldots,m} \{f_{ij} + s_{ihj} + q_{ij}\}$$

where $s_{ihj} = q_{ij} = 0$ for $j = k$.

Figure 6.4 shows how these computations are performed in procedure `Makespans()`. Steps 1, 2, and 3 of `Makespans()` take $O(km)$ time each. Step 4 is $O(k \log m)$. Therefore, this procedure is executed in $O(km)$ time. Figure 6.5 illustrates the procedure when job h is inserted at position 3 (between jobs 2 and 3) in a partial 4-job schedule.

Computational complexity: The complexity of Step 1 is $O(n \log n)$. At the k-th iteration of Step 2 (k jobs already scheduled), Step 2a takes $O(1)$, Step 2b takes $O(km)$, complexity of Step 2c is $O(k \log \lambda)$, Step 2d can be done in $O(\log \lambda)$ time, and Step 2e in $O(km)$. Thus the complexity of Step 2 at the k-th iteration is $O(km)$. This yields a time complexity of $O(mn^2)$ for one execution of `GRASP()` phase 1. Therefore, the overall phase 1 time complexity is $O(Nmn^2)$.

Example 6.1 (Example 3.1 continued)
We now illustrate the `GRASP()` construction phase with RCL cardinality limitation $\lambda = 2$.

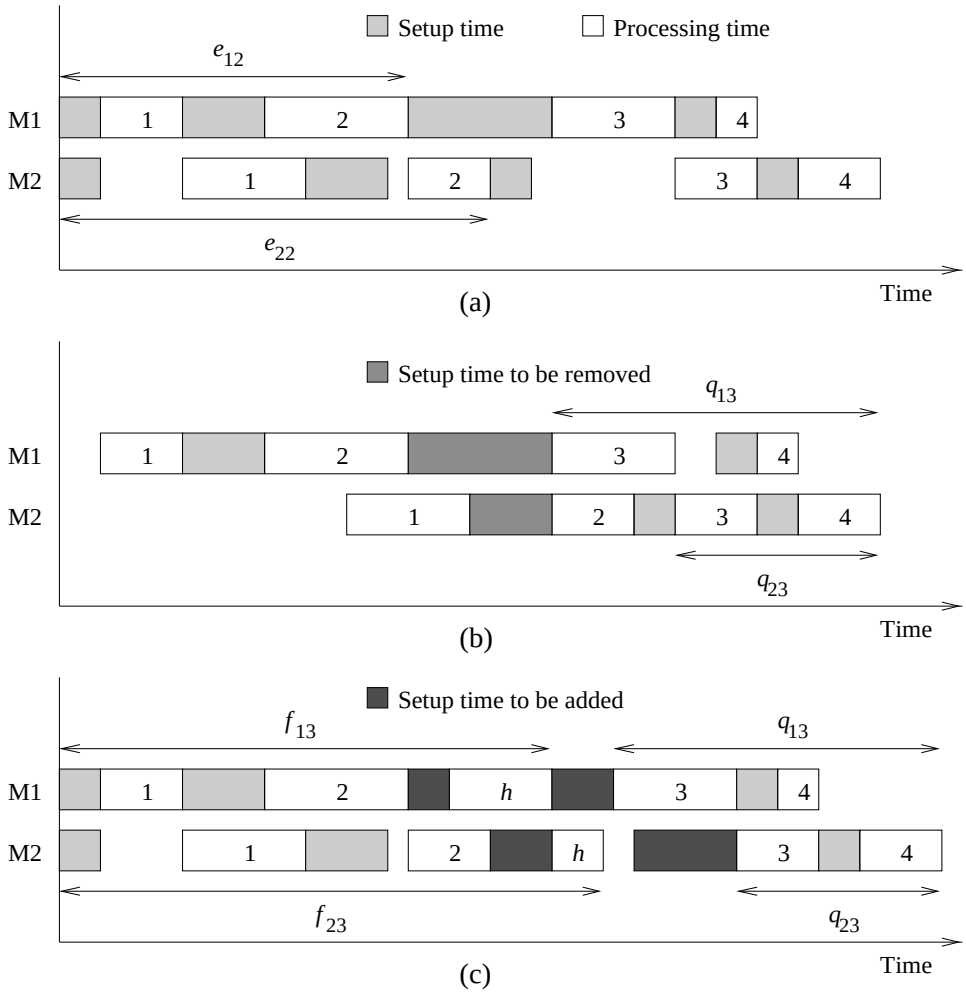


Figure 6.5: Illustration of partial makespan computation

Step 0: Initialize the set of scheduled jobs $S = \emptyset$.

Step 1: Given the total processing time for each job

j	1	2	3	4
$\sum_i p_{ij}$	8	5	6	3

form the LPT priority list as follows: $P = (1, 3, 2, 4)$.

Step 2: (Iteration 1) Job 1 is selected (and removed) from P. Now $P = (3, 2, 4)$. Since there are no scheduled jobs, insert job 1

into $S = (1)$ and go to the next iteration.

(Iteration 2) Job 3 is selected (and removed) from P. Now $P = (2, 4)$, $|S| = 1$, and $\psi(k)$ (makespan value when job 3 is inserted in position k in S) is computed as.

k	1	2
$\psi(k)$	13	18

Because $\lambda = 2$, RCL $= \{1, 2\}$. One is selected at random, say $k = 1$. Thus, job 3 is inserted in position $k = 1$ (at the beginning of S). $S = (3, 1)$.

(Iteration 3) Job 2 is selected (and removed) from P. Now $P = (4)$, $|S| = 2$, and $\psi(k)$ is computed as follows

k	1	2	3
$\psi(k)$	22	20	23

Form RCL=$\{1, 2\}$ and select one at random, say $k = 1$. Job 2 is inserted in position $k = 1$ (at the beginning of S). $S = (2, 3, 1)$.

(Iteration 4) Job 4 is selected (and removed) from P. Now $P = \emptyset$. For $|S| = 3$, $\psi(k)$ is computed as follows

k	1	2	3	4
$\psi(k)$	30	26	29	30

Form RCL $= \{2, 3\}$ and select one at random, say $k = 3$. Job 4 is inserted in position $k = 3$ (immediately succeeding job 3). $S = (2, 3, 4, 1)$.

Step 3: Output schedule $S = (2, 3, 4, 1)$ with corresponding $C_{\max}(S) = 29$.

Recall that the optimal schedule is $S^* = (3, 1, 2, 4)$ with $C_{\max}(S^*) = 24$. □

6.4 Local Search Procedures

Neighborhoods can be defined in a number of different ways, each having different computational implications. Consider, for instance, a 2-opt neighborhood definition that consists of exchanging two edges in a given tour or sequence of jobs. For this neighborhood, a move in a TSP takes $O(1)$ time to evaluate whereas a move in the SDST flowshop takes $O(mn^2)$. One of the most common neighborhoods for scheduling problems is the 2-job exchange which has been used by Widmer and Hertz [75] and by Taillard [70] for $F||C_{\max}$. We considered the 2-job exchange as well. In addition, we generalized the 1-job reinsertion neighborhood proposed by Taillard [70] for $F||C_{\max}$ to develop an L-job string reinsertion procedure. This was motivated by the presence of the sequence-dependent setup times, which suggest that subsets (or strings) of consecutive jobs might fit together in a given schedule. We tried both procedures for our problem and found that the string reinsertion uniformly outperformed the 2-job exchange, just as Taillard found the 1-job reinsertion performed better than the 2-job exchange for the regular flowshop.

6.4.1 L-Job String Reinsertion

Given a feasible schedule S, let $N_S^L(j,k)$ be the schedule formed from S by removing a string of L jobs starting at the j-th position and reinserting the string at position k. The neighborhood of S is given by

$$N(\sigma) \;=\; \left\{ N_S^L(j,k) \;:\; 1 \le j,k \le n+1 \Leftrightarrow L \right\}$$

For a given value of L, $N(S)$ is entirely defined by j and k. The size of $N(S)$ is

$$|N(S)| \;=\; (n \Leftrightarrow L)^2$$

An example of a 2-job string reinsertion neighbor is shown in Figure 6.6. The sequence on the right $S' = N_S^2(3,1)$ is formed from S by removing the 2-

job string starting at the 3-rd position (jobs 5 and 4) and reinserting it at the position 1 (immediately preceding job 2). The evaluation of all makespans can be executed in $O(n^2m)$, using the `Makespans()` procedure described in Section 6.3.

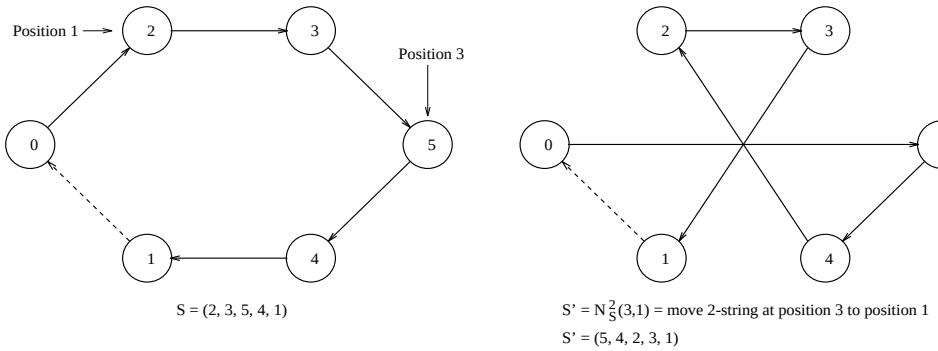


Figure 6.6: Illustration of 2-job string reinsertion neighborhood

6.4.2 Implementation Considerations

A primary concern in the implementation of local search procedures is how to move from the current feasible solution to a neighbor solution with a better objective function value. There are three fundamental ways of doing this. The first is to examine the whole neighborhood and then make a move to the "best" neighbor. The second is to examine one neighbor at a time and make a move as soon as a better solution is found. The trade-off is that in the first case we expect the incremental improvement in the objective value to be greater; however, the computational effort is higher. The third option is to examine a smaller neighborhood at the expense of the solution quality. This idea was used by Reeves in [60] for the 1-job reinsertion local search on the flowshop context. Here, we use this idea in the following way. Given a string of L jobs (typically $L \leq 4$), there are $(n \Leftrightarrow L)$ possible sites where the string can be reinserted. We observe that the evaluation of all these possible moves can be cleverly done in $O(mn^2)$, which is the same complexity of evaluating just one move. Therefore,

after making this evaluation, we make the move by reinserting the string in the best of these $(n \Leftrightarrow L)$ positions.

Heuristic	String size	NSC
`HYBRID()`	3	Lexicographic (last)
`GRASP()`	1	Lexicographic (first)

Table 6.1: Parameter selection for string reinsertion procedure

When the choice is to examine a smaller neighborhood (or subregion) as described above, we must have a criterion for selecting the "next" subregion, or in our case, how to select the next string of jobs. The neighbor selection criteria (NSC) defines a way of choosing the next subregion to be examined. Typical examples of NSC are a lexicographic strategy and a random strategy. In the former, one sorts all unexamined subregions according to a given lexicographic rule. A lexicographic first (last) rule selects the first (last) string of the sorted list and removes it from the list of unexamined strings. In a random strategy, the next string is chosen randomly among all unexamined candidates. We did a preliminary computation designed to fine-tune the local search procedure as a function of both the NSC and string size. The best choices of these parameters for a particular heuristic are shown in Table 6.1. As we can see, a string size of 1 did better in `GRASP()`, as opposed to `HYBRID()`. An explanation of this is that `GRASP()` is a heuristic that finds a feasible solution by inserting one job at a time. This produces a feasible schedule where the interrelationship among strings of jobs may not be as strong as a feasible solution delivered by `HYBRID()` which is a TSP-based heuristic. Thus, `HYBRID()` benefits better from a 3-job string reinsertion.

In general the neighborhood definition is different for each value of L; that is, a local optima with respect to $L = 1$, for instance, may not be local optima with respect to $L = 2$. Thus in practice, one can apply or combine several of these neighborhoods for different values of L, depending on the time

available to improve the solution. For instance, `HYBRID()` is a deterministic heuristic that runs very quickly. This makes a local search effort more affordable.

6.5 Experimental Evaluation

All procedures were written in C++ and run on a Sun Sparcstation 10 using the CC compiler version 2.0.1, with the optimization flag set to -O. CPU times were obtained through the C function `clock()`.

To conduct our experiments we used randomly generated data drawn from classes A, C, and D (described in Appendix E). Class D is most representative of real world instances, having a setup/processing time ratio between 20% and 40%. Classes A and C, account for a smaller (0-10%) and a larger (0-50%) ratio variation, respectively, and are intended to observe the algorithmic performance in best- and worst-case scenarios.

6.5.1 Experiment 1: Fine-Tuning Local Search for `HYBRID()`

The purpose of this experiment was to find out which local search strategy worked best within the `HYBRID()` heuristic frame. We used the L-job string reinsertion procedure (LS) in four different strategies, namely

S1: Apply LS($L = 1$)

S2: Apply LS($L = 1$) plus LS($L = 2$)

S3: Apply LS($L = 1$) plus LS($L = 2$) plus LS($L = 3$)

S4: Apply strategy 3 as many times as necessary

Strategy k, $k = 1, 2, 3$, will deliver a local optimum with respect to the N_k neighborhood. Strategy 4 delivers a local optimum with respect to all three

neighborhoods. The strategies are listed by increasing amount of computational effort. The question we want to answer is whether or not the extra effort pays off in terms of quality of the solution.

		Set D			
$m \times n$	Statistic	S1	S2	S3	S4
2×20	Number of best	4	8	14	20
	Average gap (%)	2.3	2.1	1.9	1.8
	Average time (sec)	0.9	1.3	1.5	2.4
6×20	Number of best	6	9	12	20
	Average gap (%)	9.3	9.0	8.9	8.8
	Average time (sec)	2.3	2.9	3.4	5.4
10×20	Number of best	9	11	11	20
	Average gap (%)	12.3	12.3	12.2	12.0
	Average time (sec)	3.8	4.7	5.5	8.7
2×50	Number of best	1	3	7	20
	Average gap (%)	2.2	2.0	1.9	1.8
	Average time (sec)	9.8	13.3	16.3	25.5
2×100	Number of best	1	3	7	20
	Average gap (%)	1.7	1.6	1.5	1.5
	Average time (sec)	67.9	95.5	119.8	194.0

Table 6.2: Evaluation of local search strategy for `HYBRID()` on class D instances

For a given combination of $(m \times n)$ we apply the heuristic to 20 randomly instances drawn from data classes A, C, and D. Results of this experiment are shown in Tables 6.3 and 6.2. In each cell, the table gives the number of times a given strategy found the best solution, average relative gap percentage with respect to a lower bound (described in Section 7.3), and average CPU time. The first thing to notice is that strategy S4 found in most cases more than 50% best solutions as those found by the other strategies. The performance of S4 is even better for the larger instances (in terms of the number of jobs). In terms of relative gap, the strategy 4 gives an average improvement of 0.1-0.2%. Furthermore, the Friedman test (non-parametrical test equivalent to classical

		Set A				Set C			
$m \times n$	Statistic	S1	S2	S3	S4	S1	S2	S3	S4
2×20	Number of best	1	1	7	20	8	10	14	20
	Average gap (%)	1.9	1.6	1.5	1.3	7.7	7.5	7.2	7.0
	Average time (sec)	1.4	1.9	2.3	3.8	0.4	0.6	0.7	1.2
6×20	Number of best	5	8	11	20	7	9	11	20
	Average gap (%)	6.2	6.1	6.0	5.8	3.4	2.7	16.4	15.9
	Average time (sec)	3.2	4.1	4.8	7.2	1.0	1.4	1.7	2.8
10×20	Number of best	10	10	13	20	9	12	15	20
	Average gap (%)	11.8	11.8	11.7	11.4	19.8	19.6	19.5	19.2
	Average time (sec)	5.4	6.6	7.6	11.2	1.6	2.1	2.6	4.2
2×50	Number of best	0	1	1	20	5	7	12	20
	Average gap (%)	1.3	1.1	1.0	0.8	6.2	6.1	6.0	5.9
	Average time (sec)	14.2	19.2	22.6	35.1	3.4	5.0	6.3	11.8
2×100	Number of best	0	0	0	20	3	6	10	20
	Average gap (%)	1.0	0.8	0.7	0.6	5.3	5.1	5.0	4.9
	Average time (sec)	101.8	136.9	167.6	244.5	21.2	30.6	39.3	69.6

Table 6.3: Evaluation of local search strategy for `HYBRID()` on class A and C instances

ANOVA [11]) applied to each cell finds strategy 4 to be significantly better from the statistical stand point in terms of solution quality. This improvement comes at a cost of about 50% resource usage as indicated by the CPU times. The largest average CPU time came from the 2×100 class A instances, taking about 4 minutes, which is still relatively small.

6.5.2 Experiment 2: `HYBRID()` vs. `GRASP()`

The purpose of this experiment was to evaluate the performance of both heuristics. `HYBRID()` local search strategy was set to S4 (see previous section). For `GRASP()`, we assigned $N = 100$, $K = 5$, and $\lambda = 2$. Under these settings, both heuristics use about the same amount of CPU time (`GRASP()` still is more expensive, but no more than 30% for most of the instances).

Data Set A		$n = 20$		$n = 50$		$n = 100$	
m	Statistic	H vs G		H vs G		H vs G	
2	Nbest	16	6	17	3	20	0
	Average gap (%)	1.3	1.5	0.9	1.0	0.6	0.8
	Wilcoxon test			H best		H best	
4	Nbest	10	11	14	6	17	5
	Average gap (%)	4.3	4.2	2.3	2.4	1.6	1.8
	Wilcoxon test					H best	
6	Nbest	9	13	8	13	7	13
	Average gap (%)	5.7	5.7	4.8	4.7	3.1	2.9
	Wilcoxon test						
8	Nbest	9	12	7	14	6	15
	Average gap (%)	9.5	9.5	6.5	6.2	4.8	4.5
	Wilcoxon test					G best	
10	Nbest	5	15	11	10	2	18
	Average gap (%)	11.4	11.0	7.1	6.9	6.1	5.6
	Wilcoxon test	G best				G best	

Table 6.4: Heuristic evaluation for data class A

Tables 6.4, 6.5, and 6.6 displays the results for data classes A, D, and C, respectively, in terms of the number of times a given heuristic found the best solution (Nbest) and the average relative gap (Average gap). The third line in each cell shows if any of the heuristics was found statistically better after performing the Wilcoxon test (non-paramatric pairwise test [11]) with confidence level of 99%. If the test is not significant (e.g., no heuristic is found better than the other) the cell is empty. It is observed that, for a fixed value of n, `HYBRID()` tends to do better when the number of machines is small. However, as m gets larger, then `GRASP()` tends to dominate. For example, in class D, when $n = 50$ we actually found `HYBRID()` to be statistically better than `GRASP()` for $m = 2, 4$. Then, when m takes on the values 6, 8, the Wilcoxon test does not find any heuristic better than the other one. When $m = 10$, `GRASP()` takes over. A similar behavior is observed for data classes A and C.

	Data Set D	$n = 20$		$n = 50$		$n = 100$	
m	Statistic	H vs G		H vs G		H vs G	
2	Nbest	13	8	20	0	20	0
	Average gap (%)	1.8	1.9	1.7	2.3	1.5	2.3
	Wilcoxon test			H best		H best	
4	Nbest	8	13	14	6	19	1
	Average gap (%)	6.2	5.9	4.0	4.2	3.9	4.3
	Wilcoxon test			H best		H best	
6	Nbest	8	12	11	11	18	3
	Average gap (%)	8.7	8.6	6.9	6.9	5.7	6.0
	Wilcoxon test					H best	
8	Nbest	11	9	9	11	15	5
	Average gap (%)	10.3	10.5	7.6	7.6	7.0	7.2
	Wilcoxon test					H best	
10	Nbest	11	9	5	16	14	6
	Average gap (%)	11.7	11.9	10.2	9.9	8.4	8.6
	Wilcoxon test			G best			

Table 6.5: Heuristic evaluation for data class D

When comparing the heuristic performance among the different data classes, it is observed that `GRASP()` tends to do better when setup times fluctuations are smaller. It is observed, for example, than in class A, `GRASP()` is found statistically better in 3 cases, and `HYBRID()` in 3 cases. When class D is considered, `GRASP()` is better in only one case, and `HYBRID()` in 6 cases. Finally, in class C, `HYBRID()` is found better in 10 cases, clearly dominating `GRASP()`.

6.6 Conclusions

Our computational study revealed several interesting properties about the proposed heuristics. First, it was observed that `HYBRID()` tends to perform better than `GRASP()` when the number of machines is small. Another favorable sce-

Data Set C		$n = 20$	$n = 50$	$n = 100$
m	Statistic	H vs G	H vs G	H vs G
2	Nbest	15 5	20 0	20 0
	Average gap (%)	6.9 7.4	5.8 7.6	4.7 7.3
	Wilcoxon test		H best	H best
4	Nbest	10 10	19 1	20 0
	Average gap (%)	12.3 12.5	13.1 14.6	11.8 14.4
	Wilcoxon test		H best	H best
6	Nbest	11 9	16 4	20 0
	Average gap (%)	16.5 16.6	16.2 16.9	15.7 17.7
	Wilcoxon test		H best	H best
8	Nbest	9 11	17 3	20 0
	Average gap (%)	17.7 17.6	19.1 19.8	18.3 20.3
	Wilcoxon test		H best	H best
10	Nbest	13 8	17 3	20 0
	Average gap (%)	19.1 19.2	20.8 21.6	20.6 21.9
	Wilcoxon test		H best	H best

Table 6.6: Heuristic evaluation for data class C

nario for `HYBRID()` is when the setup time fluctuations are large (data set C). This stems from the fact that the fewer the number of machines and/or the larger the magnitude of the setup times, the more the problem resembles an ATSP so a TSP-based procedure should do well. Recall that in `HYBRID()` the distance between jobs has a setup time cost component which is computed as the sum of the setup times between jobs over all the machines. In the extreme case where there is only one machine, the problem reduces entirely to an instance of the ATSP. As more machines are added, the developed cost function becomes less representative of the "distance" between the jobs.

How small does the number of machines have to be for `HYBRID()` to do better than the insertion-based heuristics depends not only on the number of jobs, but on the magnitude of the setup times as well. In data class A it was observed a threshold value of $m = 2$ and 4 for the 50-, and 100-job instances,

respectively. For class D, these threshold values increased to $m = 4$ and and 8, respectively. However, for data class C (larger setup times), `HYBRID()` was found to outperform the others with respect to both makespan (especially for the 50- and 100-job data sets) and CPU time. This implies a threshold value of $m > 10$.

Another way to explain the better performance of `HYBRID()` on the larger instances of data set C is as follows. An insertion-based heuristic (like `GRASP()`) includes a makespan estimation routine that has the setup costs as part of its performance measure; there is no other explicit treatment to the setups in the heuristic. Since the job insertion decision is made one job at a time, while the sequence-dependent setup time is dictated by the interrelationships of an entire sequence of jobs, a TSP-based heuristic tends to do better than this insertion-style method, specially when the number of machines is small when the similarities between the SDST flowshop and the ATSP are stronger.

An advantage of `GRASP()`, of course, is that by increasing the iteration counter, more and perhaps better solutions can be found. This is a trade-off that the decision maker has to evaluate under specific time constraints. In our work, we combine both heuristics into an upper bounding procedure in the exact optimization schemes described in the next chapter.

Chapter 7

Branch and Bound

7.1 Preliminaries

The feasible set of solutions of the SDST flowshop problem from a combinatorial standpoint can be represented as $X = \{\text{set of all possible } n\text{-job schedules}\}$. This is a finite set so an optimal solution can be obtained by a straightforward method that enumerates all feasible solutions in X and then outputs the one with the minimum objective value. However, complete enumeration is hardly practical because the number of cases to be considered is usually enormous. Thus any effective method must be able to detect dominated solutions so that they can be excluded from explicit consideration.

A branch-and-bound (B&B) algorithm for a minimization problem has the following general characteristics:

- a *branching rule* that defines partitions of the set of feasible solutions into subsets
- a *lower bounding rule* that provides a lower bound on the value of each solution in a subset generated by the branching rule
- a *search strategy* that selects a node from which to branch

Additional features such as *dominance rules* and *upper bounding procedures* may

also be present, and if fully exploited, could lead to substantial improvements in algorithmic performance.

A diagram representing this process is called an enumeration or search tree. In this tree, each node represents a subproblem P_i. The number of edges in the path to P_i is called the *depth* or *level* of P_i. The original problem P_0 is represented by the node at the top of the tree (root). In our case, the schedule S_0 associated with P_0 is the empty schedule.

The essentials of B&B are contained in Appendix C. The fundamentals of B&B can be found in Ibaraki [36, 37]. In this chapter we limit the discussion to our proposed algorithm, `BABAS()` (Branch-and-Bound Algorithm for Scheduling).

7.2 Branching Rule

The following branching rule is used in `BABAS()`. Nodes at level k of the search tree correspond to initial partial sequences in which jobs in the first k positions have been fixed. More formally, each node (subproblem) of the search tree can be represented by P_k, with associated schedule S_k, where $S_k = ([1], \ldots, [k])$ is an initial partial sequence of k jobs. Let U_k denote the set of unscheduled jobs. Then, for $U_k \neq \emptyset$, an immediate successor of P_k has an associated schedule of the form $S_j = ([1], \ldots, [k], j)$, where $j \in U_k$. Figure 7.1 illustrates this rule for a 4-job instance. Node P_1 represents a problem at level 1 of the enumeration tree; where only one job has been scheduled; i.e., $S_1 = (3)$.

7.3 Lower Bounds

We now develop two lower bounding procedures that turned out to be more effective than the linear programming relaxation lower bound. These procedures are based on machine completion times of partial schedules.

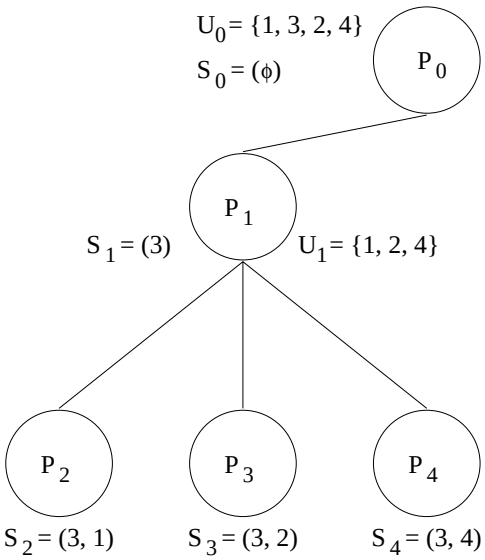


Figure 7.1: Illustration of the branching rule for a 4-job instance

Given a partial schedule S_i, let $\bar{S}_i$ denote a schedule formed by all unscheduled jobs. We shall now derive lower bounds on the value of the makespan of all possible completions $S_i\bar{S}_i$ of S_i, where $S_i\bar{S}_i$ represents the concatenation of jobs in S_i and $\bar{S}_i$. We shall be particularly concerned with the trade-off between the sharpness of a lower bound and its computational requirements. A stronger bound eliminates relatively more nodes of the search tree, but if its computational requirements become excessive, it may become advantageous to search through larger portions of the tree using a weaker bound that can be computed quickly.

7.3.1 Generalized Lower Bounds

The basic idea here is to obtain lower bounds by relaxing the capacity constraints on some machines, i.e., by assuming a subset of the machines to have infinite capacity. The only solvable case among flowshop problems is the 2-machine regular (no setups) flowshop (Johnson [39]). We know that any problem involving three or more bottleneck machines is likely to be $\mathcal{NP}$-hard.

We therefore restrict ourselves to choosing at most two machines u and v, $1 \leq u < v \leq m$, to be bottleneck machines. For any given pair (u, v) we now develop a lower bound g_{uv} by relaxing the capacity constraints on all machines except u and v. The development below shows how this lower bound can be reduced to the 2-machine case.

Let the sequence of the first k jobs be $S_k = ([1], [2], \ldots, [k])$ and the set of remaining $n \Leftrightarrow k$ (unscheduled) jobs be U_k. Given S_k, the problem of determining an optimal sequence for the remaining jobs is called a subproblem of depth k and is represented by FS(S_k). Let $\bar{S}_k = ([k+1], [k+2], \ldots, [n])$ be an arbitrary sequence of jobs in U_k, and let $p_i(U_k) = \sum_{h \in U_k} p_{ih}$. Thus the completion time $C_{i[n]}$ of job $[n]$ on machine i can be derived as follows.

$$
\begin{aligned}
C_{1[n]} &= C_{1[k]} + \sum_{h=k+1}^{n} s_{1[h-1][h]} + p_1(U_k) \\
C_{2[n]} &= \max \left\{ C_{2[k]} + \sum_{h=k+1}^{n} s_{2[h-1][h]} + p_2(U_k),\ C_{1[k]} + s_{1[k][k+1]} + T_{12}(\bar{S}_k) \right\} \\
&\ \ \vdots \\
C_{m[n]} &= \max \Bigg\{ C_{m[k]} + \sum_{h=k+1}^{n} s_{m[h-1][h]} + p_m(U_k), \\
&\qquad C_{m-1[k]} + s_{m-1[k][k+1]} + T_{m-1,m}(\bar{S}_k), \\
&\qquad \ldots,\ C_{1[k]} + s_{1[k][k+1]} + T_{1m}(\bar{S}_k) \Bigg\} \qquad (7.1)
\end{aligned}
$$

where $T_{uv}(\bar{S}_k)$ is the elapsed time from the start of job $[k+1]$ on machine u until the finish of job $[n]$ on machine v. Subproblem FS(S_k) is to determine the sequence $\bar{S}_k$ that minimizes $C_{\max}(S_k \bar{S}_k) \equiv C_{m[n]}$, the makespan of schedule $S_k \bar{S}_k$.

The definition of $T_{uv}(\bar{S}_k)$ is consistent with subsequences of $\bar{S}_k$, that is, $T_{uv}(([k+1], \ldots, [j]))$ is the elapsed time from the start of job $[k+1]$ on machine u until the finish of job $[j]$ on machine v, for $k+1 \leq j \leq n$. Thus

$T_{uv}(([k+1],\ldots,[j]))$ can be recursively computed as follows: We first initialize

$$T_{uu}(([k+1])) \;=\; p_{u[k+1]}$$

and then compute

$$T_{uw}(([k+1])) \;=\; \sum_{i=u}^{w} p_{i[k+1]}$$

for $w = u+1,\ldots,v$. Finally,

$$\begin{aligned} T_{uu}(([k+1],\ldots,[j])) &= p_{u[k+1]} + \sum_{h=k+2}^{j} s_{u[h-1][h]} + p_{u[h]} \\ T_{uw}(([k+1],\ldots,[j])) &= \max\Big\{T_{uw}(([k+1],\ldots,[j \Leftrightarrow 1])) + s_{w[j-1][j]}, \\ &\qquad T_{u,w-1}(([k+1],\ldots,[j]))\Big\} + p_{w[j]} \end{aligned}$$

for $j = k+1,\ldots,n$ and $w = u+1,\ldots,v$.

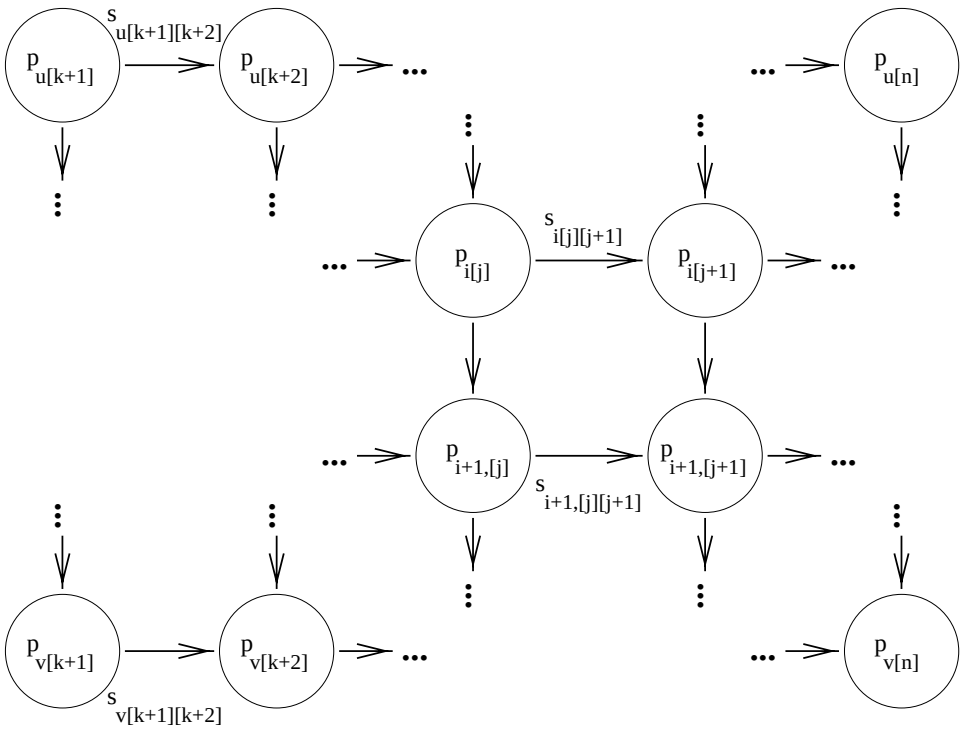


Figure 7.2: Directed graph G_{uv} for computation of T_{uv} in a SDST flwoshop

There is an alternate way to look at this recursion. To help understand the computations we introduce the following directed graph G_{uv} (depicted in Figure 7.2) which is constructed as follows: for each operation, say the processing of job $[j]$ on machine i, there is a node $(i[j])$ with a weight that is equal to $p_{i[j]}$. For each machine i, $i \in \{u, u+1, \ldots, v \Leftrightarrow 1, v\}$, there is a node $(i[k+1])$ that represents the initial or current state (job $[k+1]$ is the first job in $\bar{S}_k$). The setup times $s_{i[j][j+1]}$ are represented by an arc going from node $(i[j])$ to node $(i[j+1])$ with a weight that is equal to $s_{i[j][j+1]}$, for $i = u, u+1, \ldots, v \Leftrightarrow 1, v$, $j = k+1, \ldots, n \Leftrightarrow 1$. Node $(i[j])$, $i = u, u+1, \ldots, v \Leftrightarrow 1$, $j = k+1, \ldots, n \Leftrightarrow 1$, also has an arc going to node $(i+1, [j])$ with zero weight. Note that nodes corresponding to machine v have only one outgoing arc, and that node $(v[n])$ (target) has no outgoing arcs. The following proposition establishes the relationship between $T_{uv}(\bar{S}_k)$ and the critical path of G_{uv}.

Proposition 7.1 *$T_{uv}(\bar{S}_k)$, with $\bar{S}_k = ([k+1], \ldots, [n])$, is determined by the maximum length or critical path from node $(u[k+1])$ to node $(v[n])$.*

Proof: The proof is by induction on $w + j$ (second machine index and job index of last job in subsequence $([k+1], \ldots, [j])$. The trivial case $w + j = u + k + 1$ corresponds to $w = u$ and $j = k + 1$ and is easily verified (only source node $(u[k+1])$ involved with length $T_{uu}(([k+1]))$).

The induction hypothesis assumes that $T_{uw}(([k+1], \ldots, [j]))$ is the maximum length path from node $(u[k+1])$ to node $(w[j])$ holds for $w + j < i + l$. It remains to prove that this result holds for $w + j = i + l$ as well.

Consider $T_{ui}(([k+1], \ldots, [l]))$ given by

$$T_{ui}(([k+1], \ldots, [l])) = \max\Big\{T_{ui}(([k+1], \ldots, [l \Leftrightarrow 1])) + s_{i[l-1][l]}, T_{u,i-1}(([k+1], \ldots, [l]))\Big\} + p_{i[j]}$$

Since each of the T_{uv} in the maximization above has $w + j = i + l \Leftrightarrow 1 < i + 1$, by the induction hypothesis, those represent maximum length paths from source

node to node $(i[l \Leftrightarrow 1])$ and $(i \Leftrightarrow 1, [l])$, respectively. Since these are the only two nodes preceding node $(i[l])$, it follows that $T_{ui}(([k+1], \ldots, [l]))$ is the maximum length path from the source to node $(i[l])$ and the result is established. ■

Given the structure of G_{uv}, the length of the critical path from $(u[k+1])$ to $(v[n])$ (or equivalently, $T_{uv}(\bar{S}_k)$) is also given by

$$\begin{aligned} T_{uv}(\bar{S}_k) \;=\; \max_{k<t_u\le t_{u+1}\le\cdots\le t_{v-1}\le t_v\le n} \Bigg\{ & p_{u[k+1]} + \sum_{h=k+2}^{t_u} \left(s_{u[h-1][h]} + p_{u[h]}\right) \\ & + p_{u+1[t_u]} + \sum_{h=t_u+1}^{t_{u+1}} \left(s_{u+1[h-1][h]} + p_{u+1[h]}\right) \\ & + \ldots \\ & + p_{v-1[t_u]} + \sum_{h=t_{v-1}+1}^{t_v} \left(s_{v-1[h-1][h]} + p_{v-1[h]}\right) \\ & + p_{v[t_v]} + \sum_{h=t_v+1}^{n} \left(s_{v[h-1][h]} + p_{v[h]}\right) \Bigg\} \end{aligned} \tag{7.2}$$

for $1 \le u < v \le m$, where $\sum_{h=a}^{b}(\cdot) = 0$ for $b < a$. Thus the maximization in (7.2) consists of finding the $t_u, t_{u+1}, \ldots, t_{v-1}, t_v$ that define the critical path on G_{uv}, where t_i corresponds to the index of the job where the critical path crosses from level i to level $i+1$ on G_{uv}.

Recall that the maximization on the right-hand side of (7.2) is only used to find the T_{uv} for a given sequence $\bar{S}_k$, but in fact, the main problem is to find the subsequence $\bar{S}_k$ in U_k that minimizes $C_{m[n]}$ in (7.1). As can be seen from (7.1), minimizing $T_{uv}(\bar{S}_k)$ yields a lower bound on $C_{m[n]}$.

The minimization of $T_{uv}(\bar{S}_k)$ is as hard as the problem FS(S_k) (minimizing $C_{m[n]}$ in (7.1)), even for $T_{u,u+1}(\bar{S}_k)$. Hence we consider the minimization of the following lower bound of $T_{uv}(\bar{S}_k)$ by considering the case where $k < t_u = t_u + 1 = \ldots = t_{v-1} = t_v = t \le n$ and excluding all other terms in $T_{uv}(\bar{S}_k)$ (note that this is a valid lower bound since this special case corresponds to a path with length less than or equal to the length of the critical path), i.e.,

$$
\begin{aligned}
T_{uv}(\bar{S}_k) \;\geq\; & \max_{k<t\leq n}\Bigg\{p_{u[k+1]} + \sum_{h=k+2}^{t}\left(s_{u[h-1][h]} + p_{u[h]}\right) + p_{u+1[t]} + \ldots \\
& \qquad + p_{v-1[t]} + p_{v[t]} + \sum_{h=t_v+1}^{n}\left(s_{v[h-1][h]} + p_{v[h]}\right)\Bigg\} \\
= \;& \max_{k<t\leq n}\Bigg\{\sum_{h=k+1}^{t} p_{u[h]} + p_{u+1[t]} + \ldots + p_{v-1[t]} + \sum_{h=t}^{n} p_{v[h]} \\
& \qquad + \sum_{h=k+2}^{t} s_{u[h-1][h]} + \sum_{h=t+1}^{n} s_{v[h-1][h]}\Bigg\} \\
= \;& \max_{k<t\leq n}\Bigg\{\sum_{h=k+1}^{t} p_{u[h]} + \sum_{h=k+1}^{t} p_{u+1[h]} + \ldots + \sum_{h=k+1}^{t} p_{v-1[h]} \\
& \qquad + \sum_{h=t}^{n} p_{u+1[h]} + \ldots + \sum_{h=t}^{n} p_{v-1[h]} + \sum_{h=t}^{n} p_{v[h]} \\
& \qquad \Leftrightarrow \sum_{h=k+1}^{n} p_{u+1[h]} \Leftrightarrow \ldots \Leftrightarrow \sum_{h=k+1}^{n} p_{v-1[h]} \\
& \qquad + \sum_{h=k+2}^{t} s_{u[h-1][h]} + \sum_{h=t+1}^{n} s_{v[h-1][h]}\Bigg\} \\
= \;& \max_{k<t\leq n}\Bigg\{\sum_{h=k+1}^{t}\left(\sum_{i=u}^{v-1} p_{i[h]}\right) + \sum_{h=t}^{n}\left(\sum_{i=u+1}^{v} p_{i[h]}\right) \\
& \qquad + \sum_{h=k+2}^{t} s_{u[h-1][h]} + \sum_{h=t+1}^{n} s_{v[h-1][h]}\Bigg\} \Leftrightarrow \sum_{i=u+1}^{v-1} p_i(\bar{S}_k) \\
\geq \;& \max_{k<t\leq n}\Bigg\{\sum_{h=k+1}^{t}\left(\sum_{i=u}^{v-1} p_{i[h]}\right) + \sum_{h=t}^{n}\left(\sum_{i=u+1}^{v} p_{i[h]}\right)\Bigg\} \\
& \qquad + \sum_{h=k+2}^{n} s^{uv}_{[h-1][h]} \Leftrightarrow \sum_{i=u+1}^{v-1} p_i(\bar{S}_k)
\end{aligned}
$$

where $s^{uv}_{[h-1][h]} = \min\{s_{u[h-1][h]}, s_{v[h-1][h]}\}$. Let

$$
Z_{uv}(\bar{S}_k) \;=\; \max_{k<t\leq n}\Bigg\{\sum_{h=k+1}^{t}\left(\sum_{i=u}^{v-1} p_{i[h]}\right) + \sum_{h=t}^{n}\left(\sum_{i=u+1}^{v} p_{i[h]}\right)\Bigg\}
$$

The problem of minimizing $Z_{uv}(\bar{S}_k)$ is reduced to a solvable 2-machine flowshop

(Johnson's algorithm) with processing times

$$\begin{aligned} p'_{1j} &= \sum_{i=u}^{v-1} p_{ij} \\ p'_{2j} &= \sum_{i=u+1}^{v} p_{ij} \end{aligned}$$

Let $Z^*_{uv}(\bar{S}_k)$ be its minimum value.

The problem of minimizing $\sum_{h=k+2}^{n} s^{uv}_{[h-1][h]}$ corresponds to finding a shortest tour of an ATSP on $n \Leftrightarrow k$ vertices. Let $S^*_{uv}(\bar{S}_k)$ be a lower bound for this ATSP. Then

$$T_{uv}(\bar{S}_k) \geq Z^*_{uv}(\bar{S}_k) + S^*_{uv}(\bar{S}_k) \Leftrightarrow \sum_{i=u+1}^{v-1} p_i(\bar{S}_k) \qquad 1 \leq u < v \leq m$$

Now note the following valid lower bounds for the starting time of job $[k+1]$ on machine u

$$\begin{aligned} & C_{u[k]} + \min_{h \in U_k} \left\{ s_{u[k]h} \right\} \\ & C_{u-1[k]} + \min_{h \in U_k} \left\{ s_{u-1[k]h} + p_{u-1,h} \right\} \\ & C_{u-2[k]} + \min_{h \in U_k} \left\{ s_{u-2[k]h} + p_{u-2,h} + p_{u-1,h} \right\} \\ & \vdots \\ & C_{1[k]} + \min_{h \in U_k} \left\{ s_{1[k]h} + p_{1h} + \ldots p_{u-1,h} \right\} \end{aligned}$$

Denote by $T^{min}_{i,u-1}$ the minimum elapsed time (among all unscheduled jobs) from the finish of job $[k]$ on machine i until the finish time of job $[k+1]$ on machine $u \Leftrightarrow 1$, for $i = 1, \ldots, u$, i.e.,

$$T^{min}_{i,u-1} = \min_{h \in U_k} \left\{ s_{i[k]h} + \sum_{q=i}^{u-1} p_{qh} \right\}$$

where the case $i = u$ corresponds to $T^{min}_{u,u-1} = \min_{h \in U_k} \left\{ s_{u[k]h} \right\}$. A lower bound on the starting time of job $[k+1]$ on machine u is then given by

$$\max_{1\leq i\leq u}\left\{C_{i[k]}+T^{min}_{i,u-1}\right\}$$

Note that once the last job $[n]$ has finished on machine v, the remaining time until termination (assuming no idle time) is $\sum_{i=v+1}^{m} p_{i[n]}$. This yields the following lower bound for the elapsed time since the finish of job $[n]$ on machine v until the finish of job $[n]$ on machine m:

$$\min_{h\in U_k}\left\{\sum_{i=v+1}^{m} p_{ih}\right\}$$

We can thus establish the following generalized lower bound $g_{uv}(\bar{S}_k)$ on $C_{\max}$

$$g_{uv}(\bar{S}_k) \;=\; \max_{1\leq i\leq u}\left\{C_{i[k]}+T^{min}_{i,u-1}\right\}+Z^{*}_{uv}(\bar{S}_k)+S^{*}_{uv}(\bar{S}_k)$$

$$\Leftrightarrow \sum_{i=u+1}^{v-1} p_i(\bar{S}_k)+\min_{h\in U_k}\left\{\sum_{i=v+1}^{m} p_{ih}\right\}$$

for any $1 \leq u < v \leq m$. Note that the optimal sequence of the jobs in the embedded 2-machine flowshop (for given u, v) has to be determined only once for FS($\emptyset$), the original problem, since it does not change if some jobs are removed nor it is influenced by the fact that machine v is not available until $C_{v[k]}$.

In summary, for a given pair of machines (u, v), we have derived a generalized lower bound g_{uv} which may be computed for various machine pairs (u, v). If $W = \{(u_1, v_1), \ldots, (u_w, v_w)\}$ is a set of machine pairs, then the corresponding overall lower bound GLB(W) is defined by

$$\mathrm{GLB}(W) \;=\; \max\left\{g_{u_1,v_1},\ldots,g_{u_w,v_w}\right\}.$$

Note that there are $m(m \Leftrightarrow 1)/2$ possible pairs (u, v); however, the load for computing GLB based on all pairs is too heavy. Therefore, we only consider

the following subsets of machine pairs $W_0 = \{(1,2),(2,3),\ldots,(m \Leftrightarrow 1, m)\}$, $W_1 = \{(1,m),(2,m),\ldots,(m \Leftrightarrow 1, m)\}$, and $W_2 = W_0 \cup W_1$, which contains $O(m)$ pairs. Our empirical work (Section 7.8) has shown that GLB(W_1) provides better results than GLB(W_0) and is faster to compute than GLB(W_2).

7.3.2 Machine-Based Lower Bounds

In the previous section we developed a family of lower bounds g_{uv} for $1 \leq u < v \leq m$, based on a pair (u,v) of bottleneck machines. Consider now the case $u = v$; that is, there is only one bottleneck machine and the capacity of all other machines is relaxed. Thus it is possible to find m additional lower bounds g_u, $1 \leq u \leq m$.

Again, let the sequence of the first k jobs fixed be $S_k = ([1],[2],\ldots,[k])$ and the set of remaining $= n \Leftrightarrow k$ (unscheduled) jobs be U_k. For an arbitrary sequence of jobs in U_k, $\bar{S}_k = ([k+1],[k+2],\ldots,[n])$, let $T_u(\bar{S}_k)$ be the elapsed time from the starting time of job $[k+1]$ until the finish time of job $[n]$ on machine u. Then $T_u(\bar{S}_k)$ is given by

$$\begin{aligned} T_u(\bar{S}_k) &= p_{u[k+1]} + \sum_{h=k+2}^{n} (s_{u[h-1][h]} + p_{u[h]}) \\ &= p_u(\bar{S}_k) + \sum_{h=k+2}^{n} s_{u[h-1][h]} \end{aligned}$$

Since $p_u(\bar{S}_k)$ is constant for any sequence, the problem of minimizing $T_u(\bar{S}_k)$ corresponds to finding a sequence that minimizes $\sum_{h=k+2}^{n} s_{u[h-1][h]}$, which is equivalent to finding the shortest tour in an ATSP on $n \Leftrightarrow k$ vertices. Let $S_u^*(\bar{S}_k)$ be a lower bound for this ATSP. Then

$$g_u(\bar{S}_k) = \max_{1 \leq i \leq u} \left\{ C_{i[k]} + T_{i,u-1}^{min} \right\} + S_u^*(\bar{S}_k) + \min_{h \in U_k} \left\{ \sum_{i=u+1}^{m} p_{ih} \right\} \qquad (7.3)$$

for $1 \leq u \leq m$ is a valid lower bound on $C_{\max}$, where the first and last terms on the right-hand side are a lower bound on the starting time of job $[k+1]$ on

machine u, and a lower bound on the elapsed time between the finish of job $[n]$ on machine u and the finish of job $[n]$ on machine m, respectively, as developed in the previous section.

The fact that the setup time between jobs [k] and $[k+1]$, $s_{u[k][k+1]}$, is not considered in the computation of $T_u(\bar{S}_k)$ allows us to use the first term on the right-hand side of (7.3) as a lower bound for the starting time of job $[k+1]$ on machine u. It might be advantageous, however, to include this setup time ($s_{u[k][k+1]}$) in the computations to improve the lower bound S_u^* of the related ATSP. The trade-off is that by doing so, we no longer can use the first term on the right-hand side of (7.3). This alternate bound is expressed as

$$g'_u(\bar{S}_k) \quad = \quad L'_u(\bar{S}_k) + \min_{h \in U_k} \left\{ \sum_{i=u+1}^{m} p_{ih} \right\}$$

where L'_u is valid lower bound on $\sum_{h=k+1}^{n} s_{u[h-1][h]}$.

7.3.3 ATSP Lower Bounds

In deriving the GLB and MBLB, we have to deal with solving an ATSP at some point. The ATSP itself is an $\mathcal{NP}$-hard problem; however, since we are only interested in a lower bound, any valid lower bound for the ATSP will suffice.

In our work, we used the assignment problem (AP) lower bound, which is obtained by relaxing the connectivity (subtour elimination) constraints for the ATSP. It has been documented (Balas and Toth [4]) that the AP bound is very sharp for the ATSP. (This is not necessarily true for the symmetric TSP.)

7.4 Search Strategy

The search strategy we use selects the subproblem with the best bound; e.g., the smallest lower bound in case of a minimization problem. This approach is motivated by the observations that the subproblem with the best lower bound

has to be evaluated anyway and that it is more likely to contain the optimal solution than any other node. As shown in [36], this strategy has the characteristic that, if other parts of a branch-and-bound algorithm are not changed, the number of partial problems decomposed before termination is minimized.

Another well known strategy is depth-first search, which is mostly used in situations where it is important to find feasible solutions quickly. However, we do not consider it since feasibility is not an issue.

7.5 Dominance Rule

We now establish some conditions under which all completions of a partial schedule S_k (associated with subproblem P_k) can be eliminated because a schedule at least as good exists among the completions of another partial schedule S_j (corresponding to subproblem P_j). Let $J(S_j)$ and $J(S_k)$ denote the index sets of jobs corresponding to S_j and S_k, respectively; $l(S)$ denote the index of the last scheduled job in schedule S; and $C_i(S)$ denote the completion time of the last scheduled job in S on machine i. Then P_j dominates P_k if for any completion $S_k\bar{S}_k$ of S_k there exists a completion $S_j\bar{S}_j$ of S_j such that $C_{\max}(S_j\bar{S}_j) \leq C_{\max}(S_k\bar{S}_k)$. This is stated formally in the following theorem.

Theorem 7.1 *If $J(S_j) = J(S_k)$, $l(S_j) = l(S_k)$, and $C_i(S_j) \leq C_i(S_k)$ for all $i \in I$, then P_j dominates P_k.*

Proof: Let Q be a schedule and $q_i(Q)$ be the elapsed time between the starting of the first job in Q on machine i and the end of operations. Then for a partial schedule S_k, let Q be any schedule formed by the jobs in U_k (set of unscheduled jobs). The makespan of S_kQ can be computed as

$$C_{\max}(S_kQ) \quad = \quad \max_{i \in I} \left\{ C_i(S_k) + s_{i,l(S_k),h} + q_i(Q) \right\}$$

where h is the job index of the first job in Q. Let P_j be a subproblem such that $J(S_j) = J(S_k)$ (its corresponding partial schedule S_j has the same job indices as those of schedule S_k), $l(S_j) = l(S_k)$ (have the same job scheduled last), and $C_i(S_j) < C_i(S_k)$ for $i \in I$. Since the set of unscheduled jobs is the same for both subproblems, S_jQ is also a valid completion for P_j, and

$$s_{i,l(S_j),h} + q_i(Q) \quad = \quad s_{i,l(S_k),h} + q_i(Q) \qquad i \in I$$

Therefore

$$\begin{aligned} C_i(S_j) \leq C_i(S_k) \quad i \in I \quad &\Rightarrow \quad C_i(S_j) + s_{i,l(S_j),h} + q_i(Q) \\ &\qquad \leq C_i(S_k) + s_{i,l(S_k),h} + q_i(Q) \quad i \in I \\ &\Rightarrow \quad \max_{i \in I} \left\{ C_i(S_j) + s_{i,l(S_j),h} + q_i(Q) \right\} \\ &\qquad \leq \max_{i \in I} \left\{ C_i(S_k) + s_{i,l(S_k),h} + q_i(Q) \right\} \\ &\Rightarrow \quad C_{\max}(S_jQ) \leq C_{\max}(S_kQ) \end{aligned}$$

which shows that P_j dominates P_k. ■

A second dominance rule arises for the special case where there is no idle time between a subsequence of any three particular jobs in a schedule. This is presented in Lemma D.1 in Appendix D. Two other special cases, the first related to reversing the job sequence and the second to the independence of processing times and machines, are also discussed in Appendix D.

In terms of computational effort, determining whether a given subproblem P_k is dominated implies: (a) searching for another subproblem (at the same level), and (b) checking conditions of Theorem 7.1. Step (a) can be done in $O(\log T)$ time, where $T = O(2^d)$ is the size of search tree up to depth d (if done efficiently, there is no need to search the whole tree). Operation (b) takes $O(m)$ time. At level d, there are potentially $O(2^d)$ nodes, thus the worst-case

complexity to determine whether a given subproblem (at depth d) is dominated is $O(md2^d)$.

Despite this worst-case complexity, the implementation of this dominance rule has had a strong positive impact in the performance of `BABAS()`. Computational results are provided in Section 7.8.

7.6 Upper Bounds

It is well known that branch-and-bound computations can be reduced by using a heuristic to find a good solution to act as an upper bound prior to the application of the enumeration algorithm, as well as at certain nodes of the search tree. With this in mind we have adapted `HYBRID()` and `GRASP()` (described in Chapter 6) to handle partial schedules.

In our basic algorithm, we apply both heuristics with extensive local search at the root node to obtain a high quality feasible solution. Once the algorithm is started, an attempt is made to find a better feasible solution every time UPPER_BOUND_LOG nodes are generated, where UPPER_BOUND_LOG is a user-specified parameter. In our experiments, we set this parameter to 50. At the intermediate stages, we do not do a full local search but try to balance the computational load. Once `BABAS()` satisfies the stopping criteria, if the best feasible solution is not optimal, we apply an extensive local search to ensure that a local minimum has been obtained.

7.7 Partial Enumeration

Partial enumeration is a truncated branch-and-bound procedure similar to what is called beam search [50]. Instead of waiting to discard a portion of the tree that is *guaranteed* not to contain the optimum, we may discard parts of the tree that are not *likely* to contain the optimum. One essential is to have a good

measure of what "likely" means.

The way we handle the partial enumeration is as follows. During the branching process, every potential child is evaluated with respect to a *valuation function* h. Those potential subproblems whose valuation function do not meet a certain pre-established criterion are discarded. We implemented this idea by ranking the potential children by increasing value of h and then discarding the worst ρn nodes, where $\rho \in [0,1]$ is a user-specified parameter. The larger the value of ρ, the more nodes that will be eliminated from consideration. The case $\rho = 0$ coincides with regular branch and bound.

A Valuation Function

To develop a valuation function h we make use of the following cost function C_{jk} for each pair of jobs $j, k \in J$:

$$C_{jk} \quad = \quad \theta R_{jk} + (1 \Leftrightarrow \theta) S_{jk}$$

where $\theta \in [0,1]$ is a weight factor, R_{jk} is a term that penalizes a "bad" fit from the flowshop perspective, and S_{jk} is a term that penalizes large setup times. This cost measure was introduced in Section 6.2 where it was used to develop the TSP-based `HYBRID()` heuristic for the SDST flowshop with very good results. A detailed description on how to estimate R_{jk} and S_{jk} is given in that section.

Let P_j be the node from which branching is being considered with corresponding partial schedule S_j. Let $l(S_j)$ be the index of the last scheduled job in S_j. Then, for every $k \in U_j$, we compute $h(k) = C_{l(S_j),k}$ and then discard the worst ρn potential subproblems (in terms of $h(k)$).

Although it is likely that the nodes excluded by this procedure will not be in an optimal solution, no theoretical guarantee can be established. We should also point out the trade-off between higher confidence in the quality

of the solution and smaller computational effort when ρ is set to smaller and larger values, respectively.

7.8 Computational Experience

All routines were written in C++ and run on a Sun Sparcstation 10 using the CC compiler version 2.0.1, with the optimization flag set to -O. CPU times were obtained through the C function `clock()`.

To conduct our experiments we used randomly generated data drawn from classes A, D, and C (described in Appendix E. Recall that class D is the most representative of real world instances, having a setup/processing time ratio between 20% and 40%. Classes A and C account for a smaller (0-10%) and a larger (0-50%) ratio variation, respectively, and are intended to gauge algorithmic performance in the best- and worst-case scenarios.

7.8.1 Experiment 1: Lower Bounds

The lower bounding procedures developed in Section 7.3 were compared within the branch-and-bound enumeration framework. In our first experiment, the generalized lower bound (GLB) was evaluated for three different subsets of machine pairs.

$$\begin{aligned} W_0 &= \{(1,2),(2,3),\ldots,(m \Leftrightarrow 1, m)\} \\ W_1 &= \{(1,m),(2,m),\ldots,(m \Leftrightarrow 1, m)\} \\ W_2 &= W_0 \cup W_1 \end{aligned}$$

It is evident that GLB(W_2) will dominate the other two; however, it requires more computational effort.

Table 7.1 shows the average results for 10-job problems with machine settings $m = 4, 6$. Note that when $m = 2$, $W_0 = W_1 = W_2 = \{(1,2)\}$. The

	$m = 4$			$m = 6$		
	W_0	W_1	W_2	W_0	W_1	W_2
Average relative gap (%)	0.8	0.3	0.3	1.3	0.3	0.4
Average number of evaluated nodes (1000)	10.1	9.2	8.7	11.0	9.3	9.0
Average CPU time (min)	10.8	9.2	9.3	15.0	11.8	12.1
Optimal solutions found (%)	60	60	60	20	70	60

Table 7.1: Evaluation of GLB for 10-job class D instances

averages are taken over 10 class D instances with a stopping limit of 15 CPU minutes. The dominance rule is in effect as well. Each column shows the statistics for GLB based on W_0, W_1, and W_2, respectively. The relative gap is computed as

$$\frac{\text{best upper bound} \Leftrightarrow \text{best lower bound}}{\text{best lower bound}} \times 100\%$$

As can be seen, the quality of GLB(W_0) is inferior to the other two since a larger number of nodes has to be evaluated, resulting in larger execution times. In addition, under GLB(W_0), fewer optimal solutions are found in the allotted time (only 20% in the 6-machine instances as opposed to 60% using W_1 and W_2). When comparing GLB(W_1) and GLB(W_2), similar performance is observed in almost every statistic. In fact, GLB(W_1) was found to be slightly better than GLB(W_2). This implies that the extra effort used by GLB(W_2) (the dominant bound) is not paying off.

We now compare GLB(W_1) with MBLB (machine-based lower bound). A stopping limit of 15 CPU minutes was similarly imposed. Table 7.2 shows the average relative gap (Gap) at the start (root node) and at the end of the algorithm, percentage of times a specific procedure delivered the best lower bound (Best), and percentage of optimal solutions found (Solved) under a specific procedure, for 15-job instances of data class D. It can be seen from the table that the GLB is actually better at the root node; however, as branching

	$m = 2$		$m = 4$		$m = 6$	
	GLB(W_1)	MBLB	GLB(W_1)	MBLB	GLB(W_1)	MBLB
Gap (root) (%)	2.7	6.6	6.4	12.1	8.8	14.8
Gap (end) (%)	2.2	3.1	4.1	2.9	5.3	3.1
Best (%)	40	60	30	80	0	100
Solved (%)	30	60	0	50	0	10

Table 7.2: Lower bound comparison for 15-job class D instances

takes place, the MBLB makes more progress providing, in almost all cases, a tighter bound. There were even some instances that were solved to optimality under the MBLB alone.

One possible explanation for this result is that the MBLB, for a given machine, takes into account all the involved setup times, whereas the GLB, in its attempt to reduce the problem to a 2-machine case, loses valuable setup time information (recall that for a given machine pair (u, v), GLB uses $\min\{s_{ujk}, s_{vjk}\}$ to represent the setup time between jobs j and k). Because the MBLB procedure was uniformly better than the GLB scheme, we use it in the remainder of the experiments.

7.8.2 Experiment 2: Dominance Elimination Criterion

	$m = 2$		$m = 4$		$m = 6$	
	NDR	DR	NDR	DR	NDR	DR
Average relative gap (%)	0.7	0.0	0.0	0.0	0.1	0.0
Average number of evaluated nodes	16063	8529	5074	2985	10879	7924
Average CPU time (min)	18.3	5.8	4.8	2.3	14.2	8.4
Optimal solutions found (%)	50	100	100	100	90	100

Table 7.3: Evaluation of dominance rule for 10-job class D instances

We now evaluate the effectiveness of the dominance rule. Table 7.3 shows the average statistics over 10 class D instances for machine sizes $m =$

2, 4, 6. Each instance was run with a CPU time limit of 30 minutes and optimality gap tolerance of 0.0. The results for the algorithm with and without the dominance rule in effect are indicated by DR and NDR, respectively. As we can see, the implementation of the dominance rule has a significant impact on the overall algorithmic performance resulting in a considerably smaller number of nodes to be evaluated, and a factor of 2 reduction in CPU time. In fact, when the dominance rule was in effect, the algorithm found optimal solutions to all instances, as opposed to only 80% when the rule was not in effect.

7.8.3 Experiment 3: Partial Enumeration

	$\rho = 0$			$\rho = 0.5$			$\rho = 0.8$		
Instance	UB	Gap	Time	UB	Gap	Time	UB	Gap	Time
fs6x20.1	2022	2.8	30	2020	1.8	30	2029	1.0	1
fs6x20.2	2108	4.4	30	2111	3.2	30	2114	1.0	1
fs6x20.3	2100	5.3	30	2093	4.1	30	2106	1.0	1
fs6x20.4	1967	5.5	30	1966	3.5	30	1972	1.0	1
fs6x20.5	2095	1.5	30	2094	1.0	10	2096	1.0	1
fs6x20.6	2058	6.5	30	2057	5.3	30	2070	1.0	2
fs6x20.7	2088	5.6	30	2082	3.9	30	2088	1.0	2
fs6x20.8	2129	8.1	30	2129	6.8	30	2124	1.0	8
fs6x20.9	2106	3.7	30	2106	2.3	30	2109	1.0	1
fs6x20.10	2142	6.1	30	2130	4.2	30	2144	1.0	2

Table 7.4: Partial enumeration evaluation for 6-machine, 20-job class D instances

In this experiment, we illustrate the effect of doing partial versus complete enumeration. We ran the partial search strategy for $\rho = 0$ (normal enumeration), $\rho = 0.5$ (truncating 50% of the potential children), and $\rho = 0.8$ (truncating 80% of the potential children) for 10, 6×20 instances of data class D, with a stopping criterion of 30 minutes and relative gap fathoming tolerance of 1.0%. The overall results are displayed in Table 7.4. Results for a particular

instance are by row. For each value of ρ we tabulate upper bound (UB), relative gap percentage (Gap) and CPU time (Time) rounded to the nearest minute. It should be noted that the relative gap for the truncated versions ($\rho \in \{0.5, 0.8\}$) do not correspond to a true optimality gap, but to the best lower bound without considering the truncated nodes. As can be seen, increasing the value of ρ results in a larger number of truncated nodes, hence a quicker execution of the procedure. We can also observe that the quality of the solution decreases with the size of ρ. A good compromise seems to be around $\rho = 0.5$, but one must keep in mind that once ρ assumes a value greater than zero, the algorithm can no longer be guaranteed to provide an optimal solution to the original problem.

7.8.4 Experiment 4: `BABAS()` Overall Performance

Here we show the results when the full algorithm is applied to classes A, D and C instances of the SDST flowshop. We use the MBLB procedure, dominance elimination rule, and a relative gap fathoming tolerance of 1%. Maximum CPU time is set at 30 minutes.

Size	Optimality gap (%)			Time (sec)			Instances
$m \times n$	best	average	worst	best	average	worst	solved (%)
2×10	0.1	0.6	1.0	2	263	941	100
4	0.9	0.9	1.0	2	232	1197	100
6	0.8	0.9	1.0	3	99	529	100
2×15	0.4	0.7	1.1	3	543	1800	90
4	0.7	1.5	3.2	6	1231	1800	40
6	0.7	3.0	7.5	20	1444	1800	20
2×20	0.3	1.0	2.1	6	1083	1800	60
4	0.9	2.3	6.1	22	1473	1800	30
6	1.0	1.5	2.3	23	1445	1800	20

Table 7.5: Evaluation of `BABAS()` for class A instances

Tables 7.5, 7.6, and 7.7 display the summary statistics which were cal-

Size	Optimality gap (%)			Time (sec)			Instances
$m \times n$	best	average	worst	best	average	worst	solved (%)
2×10	0.3	0.9	1.0	1	235	560	100
4	0.8	0.9	1.0	2	68	222	100
6	0.9	1.0	1.0	29	265	450	100
2×15	0.0	1.0	2.6	3	725	1800	70
4	0.9	2.2	4.5	7	1074	1800	50
6	1.0	2.9	4.5	38	1624	1800	10
2×20	0.5	1.0	1.6	7	1298	1800	30
4	2.4	4.2	5.1	1800	1800	1800	0
6	1.5	5.0	8.1	1800	1800	1800	0

Table 7.6: Evaluation of `BABAS()` for class D instances

culated from 10 problem instances for each $m \times n$ combination for class A, D, and C, respectively. As can be seen, all 10-job instances were solved (within 1%) in an average time of less than 5 minutes, a notable improvement when compared to previous published research on this problem, where the size of the largest instances solved optimally was a 6-machine, 8-job problem. In fact, `BABAS()` was able to solve 50%, 43%, and 20% of the 15-job instances of class A, D, and C, respectively, and 37% and 23% of the 20-job instances of class A and D, respectively. Most of the instances solved corresponded to the 2-machine case. This is to be expected since the fathoming rules (lower bound and dominance) become less powerful as the number of machines increases. It was also observed that the larger the fluctuation of the setup times, the harder to solve the problem, as `BABAS()` was able to solve 62%, 56%, and 41% of all class A, D, and C instances, respectively. This stems from the fact that the setup time variation is smaller in class A, and thus finding a good sequence of jobs becomes less dependent on the setups.

Finally, Table 7.8 shows the algorithmic performance when `BABAS()` is applied to 100-job instances, respectively. The table displays best (B), average

Size	Optimality gap (%)			Time (sec)			Instances
$m \times n$	best	average	worst	best	average	worst	solved (%)
2×10	0.5	0.9	1.0	1	41	162	100
4	0.9	1.0	1.0	69	146	307	100
6	0.9	1.0	1.0	44	688	1800	100
2×15	0.9	2.1	5.7	54	944	1800	60
4	3.8	5.5	7.2	1800	1800	1800	0
6	5.0	6.8	8.3	1800	1800	1800	0
2×20	1.0	4.8	7.1	857	1706	1800	10
4	6.1	8.1	11.2	1800	1800	1800	0
6	8.3	10.9	12.2	1800	1800	1800	0

Table 7.7: Evaluation of `BABAS()` for class C instances

(A), and worst (W) optimality gaps at the start of the algorithm (root node) and at the end. Average CPU time in minutes and percentage of instances solved are shown as well. For the standard set (class D), 70% of the 2-machine instances finished with a relative gap of 1.3% or better, and the worst-case gap observed was 2.1%. For the best-case scenario (class A), 70% of the 2-machine instances were solved, and the rest had a relative gap of 1.4% or less. For the worst-case scenario (class C), 80% of the 2-machine instances ended with a relative gap of less than 5%. As expected, class C instances were harder to deal with due to the larger setup time fluctuation. In general, the average relative gap from the start to the end of the algorithm on class D instances improved, in absolute terms, by 2.0%, 0.9%, and 1.6% for the 2-, 4-, and 6-machine instances, respectively. For class A, this improvement was of 4.8%, 1.3%, and 6.5%, respectively. For class C, the improvement was of 2.2%, 3.4%, and 2.5%, respectively. We also observed that the lower bound and the dominance test were less powerful than they were in the 20 or fewer job cases. Despite of this `BABAS()` solved 70% and 20% of the 100-job class A instances with 2 and 4 machines, respectively, and 30% of the 2×100 class D instances.

Class	$m \times n$	Gap at root (%) B	A	W	Gap at end (%) B	A	W	Average time (min)	Instances solved (%)
A	2×100	1.0	5.5	15.7	0.2	0.7	1.4	28.1	70
	4	1.0	3.0	5.4	0.5	1.7	3.3	27.8	20
	6	2.1	9.2	53.3	1.3	2.7	3.7	30.0	0
D	2×100	1.2	3.4	8.4	0.6	1.4	2.1	28.1	30
	4	3.3	5.1	6.5	2.3	4.2	5.7	30.0	0
	6	5.0	7.6	9.4	4.3	6.0	7.2	30.0	0
C	2×100	5.2	6.8	8.5	4.1	4.6	5.7	30.0	0
	4	12.3	14.2	15.9	9.4	10.8	12.0	30.0	0
	6	15.3	17.6	19.3	12.3	15.1	16.6	30.0	0

Table 7.8: Evaluation of `BABAS()` for 100-job instances

7.9 Summary

We have presented and evaluated a branch-and-bound scheme for the SDST flowshop scheduling problem. Our implementation includes both lower and upper bounding procedures, and a dominance elimination criterion. The empirical results indicate the positive impact of the machine-based lower bound procedure and the dominance rule. Significantly better performance over previously published work (LP-based methods) was also obtained. We were able to solve (within 1% optimality gap) 100% of all 10-job instances tested, 50%, 43%, and 20% of the 15-job class A, D, and C instances, respectively, and 37% and 23% of the 20-job class A and D instances, respectively. In addition, for the 100-job instances, our algorithm delivered for class A instances average relative gaps of 0.7%, 1.7%, and 2.7% when applied to the 2-, 4-, and 6-machine cases, respectively. For class D, these average relative gaps were 1.4%, 4.2%, and 6.0%, respectively. `BABAS()` solved 70% and 20% of the 100-job class A instances with 2 and 4 machines, respectively, and 30% of the 2×100 class D instances.

In addition, a salient feature of our algorithm is that it permits partial

enumeration search, which can be used to obtain approximate solutions with relatively smaller computational effort.

Chapter 8

Conclusions

8.1 Summary of Research Contributions

In this work I have developed several methods to tackling one the most difficult problems in the area of machine scheduling optimization. From a practical point of view, I developed two heuristics that were found very effective in delivering high quality feasible solutions to the SDST flowshop: `HYBRID()` and `GRASP()`. `HYBRID()` attempts to exploit the embedded ATSP. To this end, it was fundamental to develop a reasonable cost measure that would assess the cost of scheduling two jobs together. The cost function I introduced accounted for the two important factors: the setup times and schedule fitness from the flowshop perspective.

`GRASP()` is a randomized procedure based on job-insertion. This idea was motivated by the fact that job-insertion heuristics had been very successful for the regular flowshop scheduling problem (no setups). I extended that idea to this problem and developed it within a GRASP framework.

In addition, a local search procedure based on shifting an L-job subsequence was developed and used in both heuristics. Empirical evidence over a large number of instances (ranging in size from from 2 machines and 20 jobs up to 10 machines and 100 jobs) drawn from different classes, showed the effectiveness of both procedures, outperforming the best existing work previously

published.

For the largest instances tested (100-job), `HYBRID()`, for example, delivered average optimality gaps of 0.6%, 1.5%, and 4.7% for 2-machine instances in classes A, D, and C, respectively.

It was also observed that `HYBRID()` performed better than `GRASP()` when the number of machines is small. Another favorable scenario for `HYBRID()` is when the setup time fluctuations are large. This stems from the fact that the fewer the number of machines and/or the larger the magnitude of the setup times, the more the problem resembles an ATSP so a TSP-based procedure should do well. Recall that in `HYBRID()` the distance between jobs has a setup time cost component which is computed as the sum of the setup times between jobs over all the machines. In the extreme case where there is only one machine, the problem reduces entirely to an instance of the ATSP. As more machines are added, the developed cost function becomes less representative of the distance between the jobs. How small does the number of machines have to be for `HYBRID()` to do better than the insertion-based heuristics depends not only on the number of jobs, but on the magnitude of the setup times as well. In data classes A and D, we observe a threshold value of $m = 2$ or 3. However, for data set C (larger setup times), `HYBRID()` was found to outperform the others with respect to both makespan (especially for the 50- and 100-job data sets) and CPU time. This implies a threshold value of $m > 10$.

Another way to explain the better performance of `HYBRID()` on the larger instances of data set C is as follows. An insertion-based heuristic (like `GRASP()`) includes a makespan estimation routine that has the setup costs as part of its performance measure; there is no other explicit treatment to the setups in the heuristic. Since the job insertion decision is made one job at a time, while the sequence-dependent setup time is dictated by the interrelationships of an entire sequence of jobs, a TSP-based heuristic tends to do better than

this insertion-style method, especially when the number of machines is small when the similarities between the SDST flowshop and the ATSP are stronger.

An advantage of `GRASP()`, of course, is that by increasing the iteration counter, more and perhaps better solutions can be found. This is a trade-off that the decision maker has to evaluate under a specific time budget. In our work, we combine both heuristics into an upper bounding procedure within the B&B enumeration scheme.

In attempting to solve the problem optimally, I investigated two different approaches. The first one was from the polyhedral perspective and was motivated by the success that polyhedral-based methods (such as B&C) have had on solving hard problems (in particular the TSP), and the strong connection between the SDST flowshop and the ATSP. I considered two different mathematical models. Model A is based in an ATSP formulation. Model B uses fewer binary variables and constraints, but its polyhedral structure had remained unexplored. I developed several polyhedral results concerning both formulations that allowed me, in turn, to develop some families of valid inequalities and to prove facial properties on several of these inequalities, for both models. These inequalities and the models were then evaluated within a B&C framework.

I found empirical evidence that using model B with B&C yields better results on solving instances of the SDST flowshop problem. However, the fact that even with the development of valid inequalities the algorithm is still unable to solve instances with 10 or more jobs shows that LP-based enumeration methods are wanting. This difficulty is inherent to the SDST flowshop (2 or more machines) since the algorithm was able to successfully solve 100- and 150-job instances restricted to the 1-machine case. Recall that minimizing the makespan in SDST flowshop is equivalent to finding the minimum length tour of an $(n+1)$-city ATSP when the number of machines is set equal to 1.

It is evident that once we start adding machines, the ATSP structure starts to weaken. One explanation for this is that, unlike the ATSP where we are looking for a good sequence of nodes, it is difficult here to characterize fully what a good sequence of jobs really is. What might be a good sequence for a certain machine, may be a bad sequence for the others. This makes this problem extremely nasty.

The quality of the LP relaxation lower bound led me to develop more efficient non-LP-based lower bounding procedures, which gave rise to a more effective enumeration scheme based on branch and bound, which was the final part of this research. This implementation included the development of both lower and upper bounding procedures, and a dominance elimination criterion. The empirical results indicate the effectiveness of the overall procedure (`BABAS()`), especially the positive impact of the machine-based lower bound procedure and the dominance rule. Significantly better performance over previously published work (LP-based methods) was also obtained. `BABAS()` was able to solve (within 1% optimality gap) 100% of all 10-job instances tested, 50%, 43%, and 20% of the 15-job class A, D, and C instances, respectively, and 37% and 23% of the 20-job class A and D instances, respectively. In addition, for the 100-job instances, `BABAS()` solved 70% and 20% of the 100-job class A instances with 2 and 4 machines, respectively, and 30% of the 2×100 class D instances. For class A, the algorithm delivered average relative gaps of 0.7%, 1.7%, and 2.7% when applied to the 2-, 4-, and 6-machine cases, respectively. For class D, the average relative gaps were 1.4%, 4.2%, and 6.0%, respectively. For class C, the average gaps were 4.6%, 10.8%, and 15.1%, respectively, which indicate how hard the problem becomes as the magnitude of the setup times increases.

A salient feature of the algorithm is that it permits partial enumeration search, which can be used to obtain approximate solutions with relatively smaller computational effort.

Another contribution of this work is the development of a random instance generator for producing several classes of instances with real-world features. This was the result of the input received from several colleagues with industrial experience related to this type of problem. The data sets have been made available to the scientific community at the Operations Research library at London College, UK, maintained by Prof. Beasley.

8.2 Directions for Future Research

There exist several avenues for research in this area. Incorporating both ready times and/or due dates/deadlines is a logical extension to the SDST flowshop that is worthwhile investigating. We should point out that most of the valid inequalities developed in this work for the SDST flowshop can actually be applied to other scheduling problems involving sequence-dependent setup times. In fact, by introducing the ready times and due dates parameters, it might be possible to develop better valid inequalities to strengthen the polyhedral set of feasible solutions. A similar situation arises in the TSP when time windows constraints are added to the model.

Another related problem is the mixed-model assembly line scheduling problem (where setup times are present). This problem, although frequently encountered in practice, has remained unexplored and presents several areas of opportunity. From the practical point of view, the development of ad hoc approximation algorithms for finding good feasible schedules is essential. On the other hand, I feel that significant progress can be made toward optimality by extending part of the research done on polyhedral theory for the SDST flowshop scheduling problem.

The SDST flowshop remains one of the hardest optimization problems in the machine scheduling field. Even though most of the medium and large sized instances were not optimally solved, our work has provided a way to find

feasible schedules with high quality, several of them with optimality gaps of less that 2%.

I am confident that this work fills the missing link between the regular flowshop manufacturing environments and all other flowshops where the setup times play an important role. While it is true that the nature of the setup times (e.g, additive setups) or a slightly different problem structure might lead to the development of ad hoc procedures in related problems, it also true that our work can certainly be applied as a first approach. I am also confident that this work will be very helpful to both practitioners and researchers dealing with the challenges of this type of machine scheduling problems and many others like it.

Appendices

Appendix A

Notation

A scheduling problem can be represented by a triplet $\alpha|\beta|\gamma$. The α field describes the machine environment and contains a single entry. The β field provides details of processing characteristics and constraints and may contain no entries, a single entry, or multiple entries. The γ field contains the objective to be minimized and usually contains a single entry. In Section A.2 we provide a definition of possible entries in each field. But first, in Section A.1, we define data associated with jobs. Most of this section is adapted from Pinedo [57].

The number of jobs is denoted by n and the number of machines by m. Both n and m are assumed to be finite. Typically, the subscripts j and k refer to jobs, whereas the subscript i refers to a machine.

A.1 Data Associated with Jobs

The following pieces of data are associated with job j.

- *Processing time* (p_{ij}). Processing time of job j on machine i. The subscript i is dropped if the processing time of job j does not depend on the machine or if job j is only to be processed on one given machine.
- *Release date* (r_j). The release date r_j of job j may also be referred to as the ready date. It is the time the job arrives at the system, that is, the

earliest time at which job j can start its processing.

- *Due date* (d_j). The due date d_j of job j represents the committed shipping or completion date (the date the job is promised to the customer). The completion of a job after its due date is allowed, but a penalty is incurred. When the due date absolutely must be met, it is referred to as a *deadline*.

- *Weight* (w_j). The weight w_j of job j is basically a priority factor, denoting the importance of job j relative to the other jobs in the system.

A.2 Problem Description

In this section, we describe possible entries for each of the fields in a triplet $\alpha|\beta|\gamma$ of a scheduling problem.

Field α. This field describes the machine environment and contains a single entry. The following examples are possible machine environments specified in the α field.

- *Single machine* (1). The case of a single machine is the simplest of all possible machine environments and is a special case of all other more complicated machine environments.

- *Flow shop* (Fm). There are m machines in series. Each job has to be processed on each one of the m machines. All jobs have the same routing, that is, they have to be processed first on machine 1, then on machine 2, and so on. After completion on one machine, a job joins the queue at the next machine. Usually, all queues are assumed to operate under the *first-in-first-out* (FIFO) discipline, that is, a job cannot "pass" another while waiting in a queue. If the FIFO discipline is in effect, the flow shop is referred to as a *permutation* flow shop and the β field includes the entry *prmu*. Often, when

a general m-machine case is considered, the m identifier may be dropped such that $F||C_{\max}$, for instance, refers to the m-machine flowshop with makespan minimization criteria.

Field β. This field provides details of processing characteristics and constraints and may contain no entries, a single entry, or multiple entries. Possible entries are:

- *Release dates* (r_j). If this symbol is present in the β field, job j may not start its processing before its release date r_j. If r_j does not appear in the β field, the processing of job j may start at any time. In contrast to the release dates, due dates are not specified in this field. The type of objective function gives sufficient indication whether there are due dates or not.
- *Sequence-dependent setup times* (s_{jk}). The s_{jk} represent the setup time between jobs j and k; s_{0k} denotes the setup time for job k if job k is first in the sequence and s_{j0} the clean-up time after job j if job j is last in the sequence (of course, s_{0k} and s_{j0} may be zero). If the setup time between jobs j and k depends on the machine, then the subscript i is included, that is, s_{ijk}. If no s_{jk} appears in the β field, all setup times are assumed to be zero or sequence independent, in which case they can simply be added to the processing times.
- *Preemptions (prmp).* Preemptions imply that it is not necessary to keep a job on a machine until completion. The scheduler is allowed to interrupt the processing of a job (preempt) at any time and put a different job on the machine. The amount of processing a preempted job already has received is not lost. When a preempted job is put back on the machine (or on another machine, in the case of machines in parallel), it only needs the machine for its *remaining* processing

time. When *prmp* is not included in the β field, preemptions are not allowed.

- *Permutation (prmu).* A constraint that may appear in the flow shop environment is that the queues in from of each machine operate according to the FIFO discipline. This implies that the order (or *permutation*) in which the jobs go through the first machine is maintained throughout the system.
- *No-wait (nwt).* The *no-wait* requirement is another phenomenon which may occur in flow shops. Jobs are not allowed to wait between two successive machines. This implies that the starting time of a job at the first machine has to be delayed to ensure that the job can go through the flow shop without having to wait for any machine. An example of such an operation is a steel-rolling mill in which a slab of steel is not allowed to wait because it would cool off. It is clear that under no-wait the machines also operate under the FIFO discipline.

Field γ. This field contains the objective to be minimized and usually contains a single entry. The objective is always a function of the completion times of the jobs, which, of course, depend on the schedule. The time job j exits the system (i.e., its completion time on the last machine on which it requires processing) is denoted by C_j. The objective may also be a function of the due dates. The *lateness* of job j is defined as

$$L_j = C_j \Leftrightarrow d_j, \tag{A.1}$$

which is positive when job j is completed late and negative when it is completed early. The *tardiness* of job j is defined as

$$T_j = \max\left(C_j \Leftrightarrow d_j, 0\right) = \max\left(L_j, 0\right). \tag{A.2}$$

The difference between tardiness and lateness lies in the fact that tardiness is never negative. The *unit penalty* of job j is defined as

$$U_j = \begin{cases} 1 & \text{if } C_j > d_j \\ 0 & \text{otherwise.} \end{cases} \tag{A.3}$$

Lateness, tardiness, and the unit penalty are the three basic due-date-related penalty functions considered in this work.

- *Makespan* ($C_{\max}$). The makespan, defined as $\max_j\{C_j\}$, is equivalent to the completion time of the last job to leave the system. A minimum makespan usually implies a high utilization of the machine(s).
- *Maximum lateness* ($L_{\max}$). The maximum lateness, defined as

  $$\max_j\{L_j\},$$

  measures the worst violation of the due dates.
- *Total weighted completion time* ($\sum w_jC_j$). The sum of the weighted completion times of n jobs gives an indication of the total holding, or inventory, costs incurred by the schedule. The sum of the completion times is in the literature often referred to as the *flow time*. The total weighted completion time is then referred to as the *weighted flow time*.
- *Total weighted tardiness* ($\sum w_jT_j$). This is also a more general cost function that the total weighted completion time.
- *Weighted number of tardy jobs* ($\sum w_jU_j$). This is not only a measure of academic interest, it is often an objective in practice as it is a measure that can be recorded very easily.

Appendix B

Polyhedral Theory Basics

The following definitions and well known theoretical results (e.g., see [53]) are used in the polyhedral study of this work (Chapter 4).

A *polyhedron* $P \subseteq R^n$ is the set of points that satisfies a finite number of linear inequalities; i.e., $P = \{x \in R^n : Ax \leq b\}$, where (A, b) is an $m \times (n+1)$ matrix. A polyhedron P is of *dimension* k, denoted $\dim(P) = k$, if the maximum number of affinely independent points in P is $k+1$. A polyhedron $P \subseteq R^n$ is *full-dimensional* if $\dim(P) = n$. Let $M = \{1, 2, \ldots, m\}$, $M^= = \{i \in M : a^i x = b_i \text{ for all } x \in P\}$ and let $M^\leq = \{i \in M : a^i x < b_i \text{ for some } x \in P\} = M \setminus M^=$. Let $(A^=, b^=)$, $(A^\leq, b^\leq)$ be the corresponding rows of (A, b), referred as the *equality* and *inequality* sets of the representation (A, b) of P. A point $x \in P$ is called an *interior point* of P is $a^i x < b_i$ for all $i \in M$.

Lemma B.1 *Let P be a polyhedron and let $(A^=, b^=)$ be its equality set. If $P \subseteq R^n$, then* $\dim(P) + \text{rank}(A^=, b^=) = n$.

Corollary B.1 *A polyhedron P is full-dimensional if and only if it has an interior point.*

The inequality $\pi x \leq \pi_0$ [or (π, π_0)] is called a *valid inequality* for P if it is satisfied by all points in P. If (π, π_0) is a valid inequality for P and $F = \{x \in P : \pi x = \pi_0\}$, F is called a *face* of P, and we say that (π, π_0)

represents F. A face F is said to be *proper* if $F \neq \emptyset$ and $F \neq P$. A face F of P is a *facet* of P if $\dim(F) = \dim(P) \Leftrightarrow 1$.

Theorem B.1 *Let* $(A^=, b^=)$ *be the equality set of* $P \subseteq R^n$ *and let* $F = \{x \in P : \pi x = \pi_0\}$ *be a proper face of* P, *where* $\pi \in R^n, \pi_0 \in R$. *Then the following two statements are equivalent:*

(i) F *is a facet of* P.

(ii) If $\lambda x = \lambda_0$ *for all* $x \in F$ *then*

$$(\lambda, \lambda_0) = (\alpha\pi + uA^=, \alpha\pi_0 + ub^=)$$

for some $\alpha \in R$ *and some* $u \in R^{|M^=|}$.

Lemma B.1 and Theorem B.1 provide two different methods of characterizing facets of a polyhedron. We will also make use use of the following results on valid inequalities for variable upper-bound flow models to develop mixed-integer cuts.

Let

$$T = \{x \in B^n, z \in R^n_+ : \sum_{j \in N^+} z_j \Leftrightarrow \sum_{j \in N^-} z_j \leq b,\ z_j \leq a_j x_j \quad \text{for } j \in N\} \quad \text{(B.1)}$$

where $N^+ \cup N^- = N$. Here $a_j \in R_+$ for $j \in N$ and $b \in R$. We say that $C \subseteq N^+$ is a *dependent set* if $\sum_{j \in C} a_j > b$.

Proposition B.1 *If* $C \subseteq N^+$ *is a dependent set,* $\lambda = \sum_{j \in C} a_j \Leftrightarrow b$, *and* $L \subseteq N^-$, *then*

$$\sum_{j \in C} [z_j + (a_j \Leftrightarrow \lambda)^+ (1 \Leftrightarrow x_j)] \quad \leq \quad b + \sum_{j \in L} \lambda x_j + \sum_{j \in N^- \setminus L} z_j \qquad \text{(B.2)}$$

is a valid inequality for T *given by (B.1).*

Appendix C

Enumerative Methods

Part of the material in this appendix is taken from Ibaraki [36, 37].

C.1 Enumeration of Solutions

The feasible region S and/or the underlying space X of many of the combinatorial optimization problems are finite sets. In such a case, an optimal solution can be obtained by a straightforward method that enumerates all feasible solutions in S and then outputs the one with the minimum (or maximum) objective value. This type of approach is called *enumeration*. A diagram representing this enumeration is called an *enumeration tree*.

However, enumeration methods may hardly be practical because the number of cases to be considered is usually enormous. Thus it becomes a major concern how to detect dominated cases so that they can be excluded from the explicit enumeration. If the exclusion is done effectively, the resulting algorithm can be fast enough to solve practical problem instances. Enumerative approaches such as branch and bound and dynamic programming are computational frameworks which make it easy to incorporate exclusion procedures.

C.2 Terminology about Directed Trees

In a directed tree, there is exactly one path from the root to each vertex v_i. The number of edges in the path is called the *depth* of v_i. The *height* of a directed tree is the largest depth of the vertices therein. If there is an edge (v_i, v_j), v_j is a *son* of v_i and v_i is the parent of v_j. Those vertices with the same parent are called *brothers*. If there is a downward path from v_i to v_j, v_j is a *descendant* of v_i and v_i is an *ancestor* of v_j. In particular v_i is an ancestor (and descendant) of itself. An ancestor (descendant) v_j of v_i, is called a *proper* ancestor (descendant) if $v_i \neq v_j$. The vertices having no sons are called *leaf vertices*.

C.3 A Branch-and-Bound Algorithm

Let P_0 be the problem to be solved. The strategy is to decompose a P_0 into a few partial problems of smaller sizes, if the given problem is too difficult or too large to attack directly. The generated partial problems should have the property that the original problem can be equivalently solved as a result of solving all of the partial problems. This *decomposition* (also called *branching operation*) may be repeatedly applied to the generated partial problems, resulting eventually in a branch-and-bound enumeration algorithm.

The above scheme of decomposition is the first step of constructing a branch-and-bound algorithm. With the branching operation only, however, the obtained algorithm is nothing but a brute force enumeration algorithm. To construct an algorithm that examines only a small portion of the entire branching tree and is still possible to provide an exact optimal solution, the following properties may be exploited.

1. If an optimal solution of a partial problem P_i is obtained by some means, it is not necessary to decompose P_i any further.

2. If it is concluded for some reason that a partial problem P_i (as well as those obtainable from P_i by branching operations) does not provide an optimal solution of P_0, P_i is said to be *fathomed* and it is not necessary to decompose P_i further.

Termination of partial problems P_i by property 1 or 2 is called the *bounding operation*. There are two basic methods to actually implement bounding operations, *lower bound test* and *dominance test*.

C.4 Branching Operations and Branching Structures

This section begins with an explanation, in a general mathematical setting, of how a branching operation is performed. The branching structure resulting from the branching operation is then defined.

C.4.1 Branching Operation

Let describe a partial problem P_i by

$$\begin{aligned} P_i: \quad & \text{minimize} \quad f(x) \\ & \text{subject to} \quad x \in S_i, \end{aligned}$$

where $S_i \subset X_i$ denotes a feasible region in the underlying space X_i.

Branching operations used in the real applications can mostly be regarded as a decomposition of set X_i into a finite number of subsets $X_{i_1}, \ldots, X_{i_k}$ such that

$$\begin{aligned} X_{i_j} &\subset X_i \qquad j = 1, 2, \ldots, k \\ \bigcup_{j=1}^{k} X_{i_j} &\supset S_i. \end{aligned} \tag{C.1}$$

This decomposition enables us to define the following k partial problems P_{i_j}, $j = 1, 2, \ldots, k$, as follows

$$\begin{array}{rll} P_{i_j}: & \text{minimize} & f(x) \\ & \text{subject to} & x \in S_{i_j}, \end{array}$$

where

$$S_{i_j} \quad = \quad S_i \bigcap X_{i_j}.$$

In most cases, the above $X_{i_1}, X_{i_2}, \ldots, X_{i_k}$ give a partition of X_i, i.e., X_{i_j} are mutually disjoint and $X_i = \bigcup_{j=1}^{k} X_{i_j}$.

C.4.2 Branching Structure

Denote the optimum value of a partial problem P_i by $f(P_i)$. If P_i is infeasible (i.e., $S_i = \emptyset$), $f(P_i) = \infty$ is assumed. $Z(P_i)$ denotes the set of optimal solutions of P_i. Generally speaking, $S_i \neq \emptyset$ does not always imply the existence of $f(P_i)$ and $Z(P_i)$ (e.g., the case of diverging to $f(P_i) = \Leftrightarrow\infty$). But such pathological cases are very exceptional for combinatorial optimization problems, and hence we shall always assume in the following discussion the existence of $f(P_i)$ and $Z(P_i)$ if $S_i \neq \emptyset$.

When P_i is decomposed into $P_{i_1}, \ldots, P_{i_k}$, by a branching operation,

$$S_i \quad = \quad \bigcup_{j=1}^{k} S_{i_j}$$

follows from condition (C.1). Thus any feasible solution $x \in S_i$ belongs to some S_{i_j} and conversely any $x \in S_{i_j}$ belongs to S_i. hence

$$\begin{aligned} f(P_i) &= \min_{1 \leq j \leq k} f(P_{i_j}) \\ Z(P_i) &= \bigcup_{j=1}^{k} \Big\{ Z(P_{i_j}) : f(P_{i_j}) = f(P_i) \Big\} \end{aligned} \qquad \text{(C.2)}$$

hold, implying that P_i can be equivalently solved by solving $P_{i_1}, \ldots, P_{i_k}$. The next property follows from (C.2) for any P_i and its sons P_{i_j}.

$$f(P_{i_j}) \quad \geq \quad f(P_i) \qquad j = 1, \ldots, k$$

This implies the following for any P_i.

$$f(P_i) \quad \geq \quad f(P_0).$$

If branching operations are applied to all generated partial problems unless it becomes meaningless, a *branching tree* results. A branching tree is a directed tree $\mathcal{B} = (\mathcal{P}, \mathcal{E})$, where $\mathcal{P}$ is a set of vertices and $\mathcal{E}$ is a set of arcs, with root $P_0 \in \mathcal{P}$. Each vertex represents a partial problem, and an arc $(P_i, P_j) \in \mathcal{E}$ shows that P_j is generated by a branching operation applied to P_i. The direction of each arc, however, is not explicitly indicated in most cases.

The resulting system $(\mathcal{B}, Z, f)$ (or $(\mathcal{B}, f)$ sometimes) is called the *branching structure* of P_0. Of course, such system is rarely given explicitly but is implicitly defined by specifying P_0 and a branching operation. Our goal is to compute $f(P_0)$ and $Z(P_0)$ (or at least one solution in $Z(P_0)$).

C.5 Lower Bounding Functions

C.5.1 General Definition

Denote a *lower bound* on the optimum value $f(P_i)$ of a partial problem P_i by $g(P_i)$, i.e.,

$$g(P_i) \quad \leq \quad f(P_i) \qquad \text{for} \qquad P_i \in \mathcal{P}. \tag{C.3}$$

When viewed as a function from $\mathcal{P}$ to R (real numbers), g is called a *lower bounding function*. Although $f(P_i)$ are usually not known, $g(P_i)$ are explicitly

computed for all generated P_i. Thus the time required for computing $g(P_i)$ is a crucial factor determining the algorithm efficiency. It is desirable to have a g which can be efficiently computed and yet provides an accurate lower bound. Such g can become available only when the structure inherent to the given problem is fully exploited.

For a partial problem P_i, a *relaxation* $\bar{P}_i$ is defined by

$$\begin{aligned} \bar{P}_i : \quad & \text{minimize} \quad g(x) \\ & \text{subject to} \quad x \in \bar{S}_i, \end{aligned}$$

where

$$\begin{aligned} S_i \subset \quad & \bar{S}_i \quad \subset X_i \\ g(x) \quad & \leq \quad f(x) \qquad \text{for} \quad x \in S_i. \end{aligned}$$

$\bar{P}_i$ has a relaxed constraint and an objective function that never exceeds the original vale $f(x)$. Thus the optimal objective value of $\bar{P}_i$, denoted by $g(P_i)$, satisfies (C.3).

The following properties are also obvious.

1. P_i is infeasible if so is $\bar{P}_i$.

2. Assume that the objective function $g(x)$ is set equal to $f(x)$. In this case, if an optimal solution of $\bar{P}_i$ is feasible in P_i, it is also an optimal solution of P_i.

In either case, P_i can be immediately terminated. The set of partial problems satisfying (1) or (2) is denoted by $\mathcal{G}$. In other words, $\mathcal{G}$ is the set of partial problems that are solved in the course of computing lower bound g.

C.5.2 Conditions on g and $\mathcal{G}$

The following five conditions are assumed throughout this thesis as general properties of g and $\mathcal{G}$.

(A) $g(P_i) \leq f(P_i)$, $P_i \in \mathcal{P}$.

(B) $g(P_i) = f(P_i)$, $P_i \in \mathcal{G}$.

(C) $g(P_i) \leq g(P_j)$ for $(P_i, P_j) \in \mathcal{E}$.

(D) $\mathcal{G} \supset \mathcal{L}$, where $\mathcal{L}$ is the set of leaf vertices in $\mathcal{B}$.

(E) $P_i \in \mathcal{G}$ implies $P_j \in \mathcal{G}$ for $(P_i, P_j) \in \mathcal{E}$.

If $\bar{P}_i$ is infeasible, it is assumed by convention that $g(P_i) = f(P_i) = \infty$. Properties (A) and (B) follow from the definition of g and $\mathcal{G}$. (C) and (E) reflect the fact that P_j is easier to handle that P_i of P_j is obtained from P_i by a decomposition. Finally, (D) comes from the fact that each leaf vertex in $\mathcal{B}$ is trivially solvable.

C.6 Upper Bounding Functions

It is sometimes easy to obtain feasible solutions to P_i, even if exact optimal solutions are difficult to compute. So-called *approximate algorithms* or *heuristic algorithms* are used for this purpose. For a minimization problem P_i, such a feasible solution provides an upper bound $u(P_i)$ on $f(P_i)$. Throughout this thesis, the following properties are assumed.

(i) $u(P_i) \geq f(P_i)$, $P_i \in \mathcal{P}$.

(ii) $u(P_i) = f(P_i)$, $P_i \in \mathcal{G}$.

Condition (ii) is reasonable since any $P_i \in \mathcal{G}$ is solved, i.e., either an optimal solution is obtained or it is concluded that P_i is infeasible. In the latter case, $u(P_i) = \infty$ may be used for convenience. In case of $u(P_i) < \infty$, it is assumed that a feasible solution of P_i realizing the upper bound $u(P_i)$ is available as a result of computing $u(P_i)$.

Upper bounds $u(P_i)$ are used to update the incumbent vale z. If good upper bounds are generated in the early stage of branch-and-bound computation, and z is thereby set to relatively small values, the lower bound test would become powerful.

It is not always assumed that $u(P_i)$ is computed for all generated P_i. $u(P_i)$ is set to ∞ if the computation is not attempted or a good bound is not found within the allotted computation time. The following notations are used for convenience.

$u = \infty$; the computation of $u(P_i)$ is not attempted for any P_i

$u = u(P_0)$; $u(P_i)$ is computed only for the initial problem P_0.

Note here that condition (ii) of $u(P_i)$ is always assumed even in these cases.

C.7 Dominance Relations

C.7.1 General Definition

The *dominance test* is another important source of bounding operations that can be as powerful as the lower bound test in some cases. It is based on a *dominance relation* D, a binary relation defined over the set of partial problems. If $P_i D P_j$ (i.e., relation D holds for an ordered pair P_i and P_j), it is said that P_i *dominates* P_j. The following properties are assumed on D.

(i) D is a partial order defined over $\mathcal{P}$.

(ii) $P_i D P_j$ implies $f(P_i) \leq f(P_j)$.

(iii) $P_i D P_j$ and $P_i \neq P_j$ imply that, for each descendant P_j' of P_j, there exists a descendant P_i' of P_i such that $P_i' D P_j'$.

(iv) During the branch-and-bound computation, no set of $(k+1)$ partial problems $P_{i_1}, \ldots, P_{i_{k+1}}, k \geq 2$, satisfying the following conditions is generated.

1. All P_{i_j} are different except that $P_{i_1} = P_{i_{k+1}}$.
2. For each $j = 1, \ldots, k$, either $P_{i_{j+1}}$ is a descendant of P_{i_j}, or $P_{i_{j+1}} D P_{i_j}$ and $f(P_{i_{j+1}}) = f(P_{i_j})$ hold.

A dominance relation is illustrated by broken arcs. Properties (iii) and (iv) are introduced to prevent a deadlock in which all P_i satisfying $f(P_i) = f(P_0)$ are terminated by the dominance test.

The partial order mentioned in condition (i) is a special binary relation defined as follows. Let R be a binary relation defined over $\mathcal{P}$. R is said to be

(A) *reflexive* if $P_i R P_j$ holds for any $P_i \in \mathcal{P}$,

(B) *symmetric* if $P_i R P_j$ implies $P_j R P_i$,

(C) *transitive* if $P_i R P_j$ and $P_j R P_k$ imply $P_i R P_k$,

(D) *antisymmetric* if $P_i R P_j$ and $P_j R P_i$ imply $P_i = P_j$.

If R has properties (A) and (C), it is called a *pseudo order*. A pseudo order is called a *partial order* if it additionally satisfies (D). Finally R is an *equivalence relation* if it satisfies properties (A), (B), and (C).

Property (ii) is a key assumption that makes the dominance test possible, i.e., P_j can be terminated if $P_i D P_j$ holds for some P_i that has already been generated. This is because an optimal solution that will be obtained from

P_i is not worse than that obtained from P_j. The dominance test is sometimes useful to exploit the problem structure in such a way that is not possible by the lower bound test.

If $f(P_i) = f(P_j)$ is concluded for $P_i \neq P_j$, either $P_i D P_j$ or $P_j D P_i$ can be used without violating properties (i)-(iv). But it is not possible to use both, in order to ensure antisymmetry of D. In this case, if P_i is tested before P_j, $P_i D P_j$ is usually chosen for the dominance test.

C.8 Branch-and-Bound Procedure

So far we have introduced the following constituents of a branch-and-bound procedure.

$(\mathcal{B}, Z, f)$: branching structure, where $\mathcal{B} = (\mathcal{P}, \mathcal{E})$ is a branching tree, $Z(P_i)$ denotes the set of optimal solutions of $P_i \in \mathcal{P}$, and $f(P_i)$ denotes the optimum value of P_i. In particular, $P_o \in \mathcal{P}$ is the original minimization problem we want to solve.

g: lower bounding function.

$\mathcal{G}$: the set of partial problems P_i, solved in the course of computing $g(P_i)$.

u: upper bounding function.

D: dominance relation.

A branch-and-bound procedure to obtain one of the optimal solutions $x \in Z(P_0)$ or all optimal solutions $Z(P_0)$ can be constructed from these. From the practical point of view, the former is more important, whereas the latter is often suitable for theoretical treatment. In this work, we are concerned with the former.

The construction of the above constituents is done as follows. Given a problem we want to solve, how to perform branching operation, and how to compute g, u, and D for the generated partial problems are first specified. These form the body of a branch-and-bound algorithm. Then for each instance P_0 of the problem, implicit application of the branching operation and computation schemes of g, u, D to all partial problems defines the above branching structure $(\mathcal{B}, Z, f)$.

C.8.1 General Description of a Branch-and-Bound Procedure

During the branch-and-bound computation, partial problems are successively generated and tested. Let $\mathcal{N}$ denote the set of partial problems currently generated. Partial problems are sometimes referred to as *vertices*, as they are represented by vertices in $\mathcal{B}$. A vertex $P_i \in \mathcal{N}$ that is neither decomposed nor tested yet is called *active*. The set of active vertices is denoted by $\mathcal{A}$. For each tested vertex in $\mathcal{N}$, its lower and upper bounds are computed. The smallest upper bound obtained so far is called the *incumbent value* and denoted by z. The solution realizing z is called the *incumbent* and stored in $\mathcal{Z}$. Upon termination, $z = f(P_o)$ holds and $\mathcal{Z}$ stores an optimal solution of P_0.

Branch-and-bound computations proceed by repeating the test of active vertices. The selection of an active vertex for the next test is done by a *search function* s, such that

$$s(\mathcal{A}) \quad \in \quad \mathcal{A}.$$

The search function s is also an important constituent that determines the overall performance, and will be discussed later. A pseudocode of procedure `branch-and-bound()` is shown in Figure C.1.

Under the assumption that branching tree $\mathcal{B}$ has a finite number of vertices and each of the steps requires finite computation time, the enumeration procedure terminates in finite computation time. See [36] for a detailed proof.

Procedure `branch-and-bound()`

Input: Problem P_0.

Output: Optimal solution x with value z.

0: *(initialization)* $\mathcal{A} \leftarrow \{P_0\}$, $\mathcal{N} \leftarrow \{P_0\}$, $z \leftarrow \infty$, and $\mathcal{Z} \leftarrow \emptyset$

1: *(search)* If $\mathcal{A} = \emptyset$, go to Step 8. Otherwise, select $P_i \leftarrow s(\mathcal{A})$ and go to Step 2.

2: *(update)* If $u(P_i) < z$, then $z \leftarrow u(P_i)$ and $\mathcal{Z} \leftarrow \{x\}$, where x is a feasible solution of P_i realizing $u(P_i) = f(x)$. Go to Step 3.

3: *($\mathcal{G}$-test)* If $P_i \in \mathcal{G}$, go to Step 7. Otherwise go to Step 4.

4: *(lower bound test)* If $g(P_i) \geq z$, go to Step 7. Otherwise go to Step 5.

5: *(dominance test)* If there exists a $P_k (\neq P_i) \in \mathcal{N}$ such that $P_k D P_i$, go to Step 7. Otherwise go to Step 6.

6: *(branch)* Decompose P_i into $P_{i_1}, \ldots, P_{i_k}$ and set

$$\mathcal{A} \leftarrow \mathcal{A} \bigcup \{P_{i_1}, \ldots, P_{i_k}\} \setminus \{P_i\},$$
$$\mathcal{N} \leftarrow \mathcal{N} \bigcup \{P_{i_1}, \ldots, P_{i_k}\}.$$

Return to Step 1.

7: *(terminate P_i)* Let $\mathcal{A} \leftarrow \mathcal{A} \setminus \{P_i\}$ and return to Step 1.

8: *(termination)* Output $x \in \mathcal{Z}$ with $z = f(x)$. Stop.

Figure C.1: Pseudocode of branch-and-bound procedure

Appendix D

Special Cases

This appendix contains three lemmas which address special cases of the SDST flowshop. The first presents a dominance rule, the second discusses the reversibility of the schedule, and the third considers specific parameter relationships. To simplify the presentation, the bracket notation for a given schedule will be dropped and we will denote a schedule S by $(1,\ldots,n)$ rather than $([1],\ldots,[n])$.

Lemma D.1 *Let $S=(1,2,\ldots,n)$ be a feasible schedule of $F|s_{ijk},\ prmu|C_{\max}$. Let e_{ij} be the earliest completion time of job j on machine i*

$$e_{ij} \;=\; \max\{e_{i-1,j},\ e_{i,j-1}+s_{i,j-1,j}\}+p_{ij}$$

for $i=1,2,\ldots,m,\ j=1,2,\ldots,n,$ and $e_{i0}=e_{0j}=0$. Let q_{ij} be the minimum remaining time from the start of job j on machine i to the end of operations on the last machine

$$q_{ij}=\max\{q_{i+1,j},\ q_{i,j+1}+s_{i,j,j+1}\}+p_{ij}$$

for $i=m,m\Leftrightarrow 1,\ldots,1,\ j=n,n\Leftrightarrow 1,\ldots,1,$ and $q_{i,n+1}=q_{m+1,j}=0$. Let j and $j+1$ be any two adjacent jobs in S ($j=1,2,...,n\Leftrightarrow 1$) and let

$$S'=(1,\ldots,j\Leftrightarrow 1,j+1,j,j+2,\ldots,n)$$

be the schedule where jobs j and $j+1$ are exchanged (with completion time e'_{ij} and remaining time q'_{ij}).

If all of the following conditions hold for each $i = 1, 2, \ldots, m$

(a) $e_{ij} = e_{i,j-1} + s_{i,j-1,j} + p_{ij}$ *(there is no idle time between jobs $j \Leftrightarrow 1$ and j in S)*

(b) $q_{i,j+1} = q_{i,j+2} + s_{i,j+1,j+2} + p_{i,j+1}$ *(there is no idle time between jobs $j+1$ and $j+2$ in S)*

(c) $e'_{i,j+1} = e'_{i,j-1} + s_{i,j-1,j+1} + p_{i,j+1}$ *(there is no idle time between jobs $j \Leftrightarrow 1$ and $j+1$ in S')*

(d) $q'_{i,j} = q'_{i,j+2} + s_{i,j,j+2} + p_{i,j}$ *(there is no idle time between jobs j and $j+2$ in S')*

(e) $s_{i,j-1,j} + s_{i,j,j+1} + s_{i,j+1,j+2} > s_{i,j-1,j+1} + s_{i,j+1,j} + s_{i,j,j+2}$

then S' has a lower makespan than S,

$$C_{\max}(S') < C_{\max}(S).$$

Proof: First notice that both S and S' are identical sequences except for jobs j and $j+1$. This implies that $e_{ik} = e'_{ik}$ for all $k = 1, 2, \ldots, j \Leftrightarrow 1$ and $q_{ik} = q'_{ik}$ for all $k = j+2, j+3, \ldots, n$. Thus, from (e) we obtain

$$\begin{aligned} e_{i,j-1} + s_{i,j-1,j} + p_{ij} + q_{i,j+2} + s_{i,j+1,j+2} + p_{i,j+1} \quad > \quad & e'_{i,j-1} + s_{i,j-1,j+1} + p_{i,j+1} \\ & + q'_{i,j+2} + s_{i,j,j+2} + p_{ij} \end{aligned}$$

for all i. Conditions (a)-(d) yield

$$e_{ij} + s_{i,j,j+1} + q_{i,j+1} > e'_{i,j+1} + s_{i,j+1,j} + q'_{ij} \quad \text{for all } i$$

In particular, this is valid for the maximum over i

$$\max_i \{e_{ij} + s_{i,j,j+1} + q_{i,j+1}\} > \max_i \left\{e'_{i,j+1} + s_{i,j+1,j} + q'_{ij}\right\}$$

But these expressions correspond to the makespan values of S and S', respectively. That is,

$$C_{\max}(S) > C_{\max}(S').$$

■

An appropriate data structure should keep track of both e_{ij} and q_{ij} for all i and j. This would make it possible to check conditions (a)-(d) in $O(m)$ time.

As seen in Section 7.3, Proposition 7.1, $T_{uv}(\bar{S}_k)$ (the elapsed time between the first job in $\bar{S}_k$ on machine u and the last job in $\bar{S}_k$ on machine v) can be computed by finding the critical path on graph G_{uv} (Figure 7.2). Note that $T_{1m}(S)$ is an equivalent form to express the makespan of schedule S, which implies, by Proposition 7.1, that its makespan is given by the critical path from node $(1,0)$ to node (m,n) in graph G_{1m}.

An interesting property can be obtained when comparing two instances of the SDST flowshop with no initial setup times. Let FS be an instance of $F|s_{ijk},\ prmu|C_{\max}$ with processing times p_{ij} and setup times s_{ijk}. Let us assume that $s_{i0k} = 0$ for all $i \in I$, and $k \in J$. Let FS$'$ be another instance of the SDST flowshop with processing and setup times given by

$$\begin{aligned} p'_{ij} &= p_{m+1-i,j}, \quad \text{and} \\ s'_{ijk} &= s_{m+1-i,k,j}, \end{aligned}$$

respectively. This basically implies that the first machine in the FS$'$ is identical to the last machine in FS; the second machine in FS$'$ is identical to machine $m \Leftrightarrow 1$ in FS, and so on. The following lemma applies to these two flowshops.

Lemma D.2 *Let $S = (1, \ldots, n)$ be a sequence of jobs in FS with corresponding makespan $C_{\max}(S)$. If the jobs in FS$'$ follow the sequence $S' = (n, n \Leftrightarrow 1, \ldots, 1)$ (with makespan $C'_{\max}(S')$), then*

$$C_{\max}(S) = C'_{\max}(S').$$

Proof: Let $S = (1, \ldots, n)$ be a feasible sequence in FS. Then its makespan $C_{\max}(S)$ is given by $T_{1m}(S)$, the length of the critical path in G_{1m}. Let G'_{1m} be the graph associated to FS$'$ under sequence $S' = (n, \ldots, 1)$. By definition of FS$'$, G'_{1m} is obtained from G_{1m} by reversing the sense of all the arcs in G_{1m}. Since the length of the critical path from does not change, it follows that $T_{1m}(S) = T'_{1m}(S')$, where $T'_{1m}(S')$ is the length of the critical path in G'_{1m}, and the proof is complete. ■

Lemma D.2 states the following reversibility result: the makespan does not change if the jobs go through the flowshop in the opposite direction in the reverse order.

Another special case of $F|s_{ijk},\ prmu|C_{\max}$ which is of interest is the so-called proportionate flowshop. In this flowshop the processing times of job j on each machine are equal to p_j, that is, $p_{ij} = p_j$, $i = 1, \ldots, m$. Minimizing the makespan in a proportionate permutation flowshop is denoted by $F|p_{ij} = p_j,\ prmu|C_{\max}$. This problem has a very special property when all setup times are equal to a constant $s_{ijk} = s$.

Lemma D.3 *For $F|p_{ij} = p_j,\ s_{ijk} = s,\ prmu|C_{\max}$, the makespan is given by*

$$C_{\max} \ = \ \sum_{j=1}^{n} p_j + ns + (m \Leftrightarrow 1) \max_j \{p_j\}$$

and is independent of the schedule.

Proof: From Figure 7.2 we can see that for any sequence of jobs $S = (1, 2, \ldots, n)$ the critical path starts at node $(1, 0)$, stays on machine 1 until it

reaches node $(1,k)$, where $k = \arg\max_j\{p_j\}$, stays on job k until it reaches node (m,k), and ends by reaching node (m,n). ■

Similar results on reversibility and proportionate flowshops for $F||C_{\max}$ are discussed in [57].

Appendix E

Data Sets

E.1 Background

Although the SDST flowshop scheduling problem has been studied in the literature using exact and heuristic methods, a common comparison base is missing. This part of my research focuses on how to randomly generate instances with real-world attributes.

According to literature, and researchers with experience with this type of problem, one of the key issues is the relationship between the setup times and the processing times. For most real-world instances this setup/processing time ratio lies between 20% and 40% (class D below). In addition, we also consider the extreme cases where the setup times are allowed both a smaller (class A) and a larger variation (class C).

	p_{ij}	s_{ijk}
Class A	$[10, 100]$	$[1, 10]$
Class C	$[50, 100]$	$[1, 50]$
Class D	$[20, 100]$	$[20, 40]$

Table E.1: Data class attributes

Table E.1 shows the different classes of data sets considered. Both processing and setup times are randomly generated according to a uniform distribution in the shown interval. As it was found in our research, solution

attempts increase in difficulty with the magnitude of the setup times. Thus, in a sense, class A (C) represent a best (worst) case scenario for our solution procedures. Most of our work, tough, is based in class D, which is the most representative of real data. Within, each data class, though, several combinations of $m \times n$ were generated with m ranging from 2 to 10 machines, and n from 10 to 100 jobs. The following sections in this chapter describe in detail the SDST flowshop random instance generator.

The code for the random generators and all data instances are available at the following world-wide web sites:

- Operations Research library at the Imperial College, United Kingdom, maintained by Prof. J. Beasley:
  `http://mscmga.ms.ic.ac.uk/info.html`
- Author's personal site:
  `http://www.me.utexas.edu/~roger/Pro/`

E.2 Uniform Pseudorandom Number Generator

The problem instances presented in this work are randomly generated according to the congruential generator which is based on the recursive formula:

$$X_{i+1} = (16807 X_i) \bmod (2^{31} \Leftrightarrow 1)$$

This random number generator is proposed in Bratley et al. [8] and has been used by Taillard [71] to generate random instance of several multiple machine scheduling problems such as flow shops, job shops, and open shops.

The implementation uses only 32-bit integers and provides a uniformly distribution sequence of numbers in the (0,1) interval. A pseudocode of the procedure is shown in Figure E.1.

Procedure `random()`

Input: A seed value X_0 $(0 < X_0 < 2^{31} \Leftrightarrow 1)$.

Output: A random number in (0,1) and a modified seed value X_1.

0: Initialize constants:

$a = 16,807,\ b = 127,773,\ c = 2,836,\ m = 2^{31} \Leftrightarrow 1.$

1: Modify seed:

$k = \lfloor X_0/b \rfloor$

$X_1 = a(X_0 \bmod b) \Leftrightarrow kc$

if $X_1 < 0$, then let $X_1 = X_1 + m$.

2: Output X_1/m and X_1

3: Stop

Figure E.1: Pseudocode of random number generator

Let X $(0 < X < 1)$ be a random number generated by procedure `random()` and let a and b be any two integer numbers. Then a pseudorandom number in the interval $[a, b]$ is obtained by

$$\lfloor a + X(b \Leftrightarrow a + 1) \rfloor$$

and every integer between a and b has the "same" probability of being chosen.

E.3 Flow Shop Instance Generator

This is, in a sense, an attempt not only to create instances for testing our procedures but to provide a valid set of instances that can be used as benchmarks for other researchers as well. Our random instance generator (written in C++) is available upon request. Below is a description of the flowshop generator and

the input data file that must be provided to the generator.

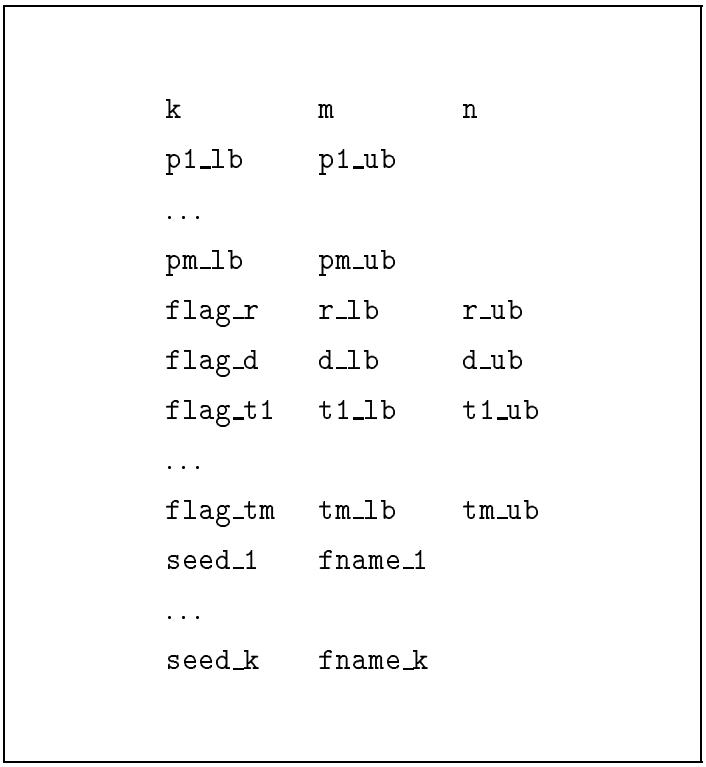


Figure E.2: Format of input file to random instance generator

The format of the input file to the generator is shown in Figure E.2. The first line contains the number of instances to be generated (`k`), the number of machines (`m`), and the number of jobs (`n`) in each instance. The lines that follow are self-explanatory. They contain the range of the uniform distribution (lower and upper bound) for the processing times, job ready times, job due dates, and machine setup times.

Notice that the generator allows enough flexibility so as to generate a different distribution for each machine for processing and the setup times. The `flag_*` parameters indicate whether such a feature should or should not be included in the data set. A value of 0 for `flag_r`, for instance, would indicate that no job ready times are generated (in such a case the values `r_lb` and

`r_ub` are not taken into account), whereas a value of 1 would indicate that job ready times will be generated from a uniform distribution in the [`r_lb, r_ub`] interval. Each of the last `k` lines, which correspond to each of the `k` instances to be generated, show both an initial random seed and a file name for the result problem.

The pseudocode for the random instance generator is outlined in Figure E.3. The procedure basically generates the processing times, the job ready times (if requested), the job due dates (if requested), and the machine setup times (if requested).

The instance generator creates `q` problems, each stored in a file (given by `fname_k`) with the following format:

- First line: random seed
- Second line: number of machines and number of jobs
- Third line: job release time flag, job due dates flag, and machine setup time flag
- Next m lines: processing times
- Next line: job ready times (only if `flag_r` $= 1$)
- Next line: job due dates (only if `flag_r` $= 1$)
- Remaining lines: machine setup times. For each of the m machines there are $n + 2$ lines. The first one contains a machine index ($0, 1, \ldots, m \Leftrightarrow 1$), the other $n + 1$ lines contain a square matrix of order $n + 1$, where entry j, k is the corresponding processing time s_{ijk}. Here, the $(n + 1)$-th row contains the initial setup times for each job and the $(n + 1)$-th column contain the finishing setup time for each job. A value of $\Leftrightarrow$1 appears along the diagonal.

See Figure E.4 for an example of a 2-machine, 3-job problem with due dates, setup times, and no release times.

Procedure `random_instance()`

Input: An input file as specified in Figure E.2.

Output: q random instances as specified in Figure E.4.

1: Read data
2: for $k = 1$ to q do
    Generate processing times
    for $i = 1$ to m do
        generate p_{ij} in [`pi_lb, pi_ub`]
    if (`flag_r = YES`)
        generate r_j in [`r_lb, r_ub`]
    if (`flag_d = YES`)
        generate d_j in [`d_lb, d_ub`]
    Generate setup times
    for $i = 1$ to m do
        if (`flag_ti = YES`)
            generate s_{ijk} in [`ti_lb, ti_ub`]
    Output data to file `fname_k`
3: Stop

Figure E.3: Pseudocode of random instance generator

```
2345671345
2    3
0    1    1
23   45   35
11   37   28
65   47   53
0
-1   14   23   00
16   -1   28   00
14   17   -1   00
09   21   19   -1
1
-1   24   03   00
26   -1   08   00
24   27   -1   00
19   01   29   -1
```

Figure E.4: Format of output file to random instance generator

Bibliography

[1] E. Balas. The asymmetric assignment problem and some new facets of the traveling salesman polytope on a directed graph. *SIAM Journal on Discrete Mathematics*, 2(4):425–451, 1989.

[2] E. Balas and M. Fischetti. The fixed-outdegree 1-arborescence polytope. *Mathematics of Operations Research*, 17(4):1001–1018, 1992.

[3] E. Balas and M. Fischetti. A lifting procedure for the asymmetric traveling salesman polytope and a large new class of facets. *Mathematical Programming*, 58(3):325–352, 1993.

[4] E. Balas and P. Toth. Branch and bound methods. In E. L. Lawler, J. K. Lenstra, A. H. G. Rinnoy Kan, and D. B. Shmoys, editors, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, pages 361–401. John Wiley & Sons, Chichester, 1990.

[5] L. Bianco, S. Ricciardelli, G. Rinaldi, and A. Sassano. Scheduling tasks with sequence-dependent processing times. *Naval Research Logistics Quarterly*, 35(2):177–184, 1988.

[6] J. Blazewicz, M. Dror, and J. Weglarz. Mathematical programming formulations for machine scheduling: A survey. *European Journal of Operational Research*, 51(3):283–300, 1991.

[7] J. Blazewicz, G. Finke, R. Haupt, and G. Schmidt. New trends in machine scheduling. *European Journal of Operational Research*, 37(3):303–317, 1988.

[8] P. Bratley, B. L. Fox, and L. E. Schrage. *A Guide to Simulation*. Springer-Verlag, New York, 1983.

[9] H. G. Campbell, R. A. Dudek, and M. L. Smith. A heuristic algorithm for the n job, m machine sequencing problem. *Management Science*, 16(10):B630–B637, 1970.

[10] J. Carlier and I. Rebai. Two branch and bound algorithms for the permutation flow shop problem. *European Journal of Operational Research*, 90(2):238–251, 1996.

[11] W. Conover. *Practical Nonparametric Statistics*. John Wiley & Sons, New York, 1980.

[12] B. D. Corwin and A. O. Esogbue. Two machine flow shop scheduling problems with sequence dependent setup times: A dynamic programming approach. *Naval Research Logistics Quarterly*, 21(3):515–524, 1974.

[13] CPLEX Optimization, Inc., Incline Village, NV. *Using the CPLEX Callable Library, Version 4.0*, 1995.

[14] H. Crowder and M. W. Padberg. Solving large-scale asymmetric traveling salesman problems to optimality. *Management Science*, 26(5):495–509, 1980.

[15] D. G. Dannenbring. An evaluation of flow shop sequencing heuristics. *Management Science*, 23(11):1174–1175, 1977.

[16] R. D. Dear. The dynamic scheduling of aircraft in the near terminal area. FTL Report R76-9, Massachussetts Institute of Technology, September 1976.

[17] F. Della Croce, V. Narayan, and R. Tadei. Two-machine total completion time flow shop problem. *European Journal of Operational Research*, 90(2):227–237, 1996.

[18] T. A. Feo and J. F. Bard. Flight scheduling and maintenance base planning. *Management Science*, 35(12):1415–1432, 1989.

[19] T. A. Feo, J. F. Bard, and K. Venkatraman. A GRASP for a difficult single machine scheduling problem. *Computers & Operations Research*, 18(8):635–643, 1991.

[20] T. A. Feo and J. L. González-Velarde. The intermodal assignment problem: Models, algorithms, and heuristics. Technical Report ORP90-10, Operations Research Program, University of Texas at Austin, August 1990.

[21] T. A. Feo and M. G. C. Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8(2):67–71, 1989.

[22] T. A. Feo and M. G. C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[23] T. A. Feo, M. G. C. Resende, and S. H. Smith. A greedy randomized adaptive search procedure for maximum independent sets. *Operations Research*, 42(5):860–878, 1994.

[24] M. Fischetti. Facets of the asymmetric traveling salesman polytope. *Mathematics of Operations Research*, 16(1):42–56, 1991.

[25] M. Fischetti and P. Toth. A polyhedral approach for the exact solution of hard ATSP instances. *Management Science*, 1997. Forthcoming.

[26] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *SIAM Journal on Applied Mathematics*, 9:551–570, 1961.

[27] N. Grötschell and M. W. Padberg. Polyhedral theory. In E. L. Lawler, J. K. Lenstra, A. H. G. Rinnoy Kan, and D. B. Shmoys, editors, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, pages 251–305. John Wiley & Sons, Chichester, 1985.

[28] J. N. D. Gupta. A functional heuristic algorithm for the flowshop scheduling problem. *Operational Research Quarterly*, 22(1):39–47, 1971.

[29] J. N. D. Gupta. Flowshop schedules with sequence dependent setup times. *Journal of the Operations Research Society of Japan*, 29(3):206–219, 1986.

[30] J. N. D. Gupta and W. P. Darrow. The two-machine sequence dependent flowshop scheduling problem. *European Journal of Operational Research*, 24(3):439–446, 1986.

[31] J. N. D. Gupta, J. C. Ho, and J. A. A. van der Veen. Single machine bicriteria scheduling with customer orders and multiple job classes. Working Paper Series 1, The Netherlands Business School, Nijenrode University, June 1994.

[32] S. K. Gupta. n jobs and m machines job-shop problems with sequence-dependent set-up times. *International Journal of Production Research*, 20(5):643–656, 1982.

[33] K. W. Hansmann. Application of new heuristics to scheduling with sequence-dependent setup times. INFORMS National Meeting, New Orleans, October 1995.

[34] J. C. Ho and Y.-L. Chang. A new heuristic for the n-job, m-machine flowshop problem. *European Journal of Operational Research*, 52(2):194–202, 1991.

[35] T. S. Hundal and J. Rajgopal. An extension of Palmer's heuristic for the flow shop scheduling problem. *International Journal of Production Research*, 26(6):1119–1124, 1988.

[36] T. Ibaraki. Enumerative approaches to combinatorial optimization: Part I. *Annals of Operations Research*, 10(1–4):1–340, 1987.

[37] T. Ibaraki. Enumerative approaches to combinatorial optimization: Part II. *Annals of Operations Research*, 11(1–4):341–602, 1987.

[38] E. Ignall and L. Schrage. Application of the branch and bound technique to some flow-shop scheduling problems. *Operations Research*, 13(3):400–412, 1965.

[39] S. M. Johnson. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, 1(1):61–68, 1954.

[40] J. Klincewicz. Avoiding local optima in the p-hub location problem using tabu search and grasp. Technical Report, AT&T Bell Laboratories, June 1989.

[41] G. Kontoravdis and J. F. Bard. A randomized adaptive search procedure for the vehicle routing problem with time windows. *ORSA Journal on Computing*, 7(1):10–23, 1995.

[42] B. J. Lageweg, J. K. Lenstra, and A. H. G. Rinnooy Kan. A general bounding scheme for the permutation flow-shop problem. *Operations Research*, 26(1):53–67, 1978.

[43] M. Laguna and J. L. González-Velarde. A search heuristic for just-in-time scheduling in parallel machines. *Journal of Intelligent Manufacturing*, 2:253–260, 1991.

[44] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. Shmoys. Sequencing and scheduling: Algorithms and complexity. In S. S. Graves, A. H. G. Rinnooy Kan, and P. Zipkin, editors, *Handbook in Operations Research and Management Science, Vol. 4: Logistics of Production and Inventory*, pages 445–522. North-Holland, New York, 1993.

[45] Y. Li, P. M. Pardalos, and M. G. C. Resende. A greedy randomized adaptive search procedure for the quadratic assignment problem. In P. M. Pardalos and H. Wolkowicz, editors, *Quadratic Assignment and Related Problems*, pages 237–261. American Mathematical Society, 1994.

[46] A. G. Lockett and A. P. Muhlemann. A scheduling problem involving sequence dependent changeover times. *Operations Research*, 20(4):895–902, 1972.

[47] Z. A. Lomnicki. A "branch-and-bound" algorithm for the exact solution of the three-machine scheduling problem. *Operational Research Quarterly*, 16(1):89–100, 1965.

[48] T. Mavridou, P. M. Pardalos, L. S. Pitsoulis, and M. G. C. Resende. A GRASP for the biquadratic assignment problem. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, 1995.

[49] K. Morizawa, H. Nagasawa, and N. Nishiyama. Complex random sample scheduling and its application to an $N/M/F/F_{\max}$ problem. *Computers & Industrial Engineering*, 27(1–4):23–26, 1994.

[50] T. E. Morton and D. W. Pentico. *Heuristic Scheduling Systems.* John Wiley & Sons, New York, 1993.

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

[52] G. L. Nemhauser, M. W. P. Savelsbergh, and G. C. Sigismondi. MINTO, a Mixed INTeger Optimizer. *Operations Research Letters*, 15:48–59, 1994.

[53] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, New York, 1988.

[54] E. Nowicki and C. Smutnicki. A fast tabu search algorithm for the flow shop problem. Report 8/94, Institute of Engineering Cybernetics, Technical University of Wrocław, 1994.

[55] M. Padberg and G. Rinaldi. An efficient algorithm for the minimum capacity cut problem. *Mathematical Programming*, 47(1):19–36, 1990.

[56] D. S. Palmer. Sequencing jobs through a multi-stage process in the minimum total time – a quick method of obtaining near optimum. *Operational Research Quarterly*, 16(1):101–107, 1965.

[57] M. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

[58] C. N. Potts. An adaptive branching rule for the permutation flow-shop problem. *European Journal of Operational Research*, 5(1):19–25, 1980.

[59] M. Queyranne and Y. Wang. Symmetric inequalities and their composition for asymmetric travelling salesman polytopes. *Mathematics of Operations Research*, 20(4):838–863, 1995.

[60] C. R. Reeves. Improving the efficiency of tabu search for machine sequencing problems. *Journal of the Operational Research Society*, 44(4):375–382, 1993.

[61] M. G. C. Resende and C. C. Ribeiro. A GRASP for graph planarization. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, 1995.

[62] R. Z. Ríos-Mercado and J. F. Bard. Heuristics for the flow line problem with setup costs. *European Journal of Operational Research*, 1997. Forthcoming.

[63] S. Sarin and M. Lefoka. Scheduling heuristics for the n-job m-machine flow shop. *OMEGA The International Journal of Management Science*, 21(2):229–234, 1993.

[64] M. W. P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6(4):445–454, 1994.

[65] J. V. Simons Jr. Heuristics in flow shop scheduling with sequence dependent setup times. *OMEGA The International Journal of Management Science*, 20(2):215–225, 1992.

[66] B. N. Srikar and S. Ghosh. A MILP model for the n-job, m-stage flowshop with sequence dependent set-up times. *International Journal of Production Research*, 24(6):1459–1474, 1986.

[67] E. F. Stafford and F. T. Tseng. On the Srikar-Ghosh MILP model for the $N \times M$ SDST flowshop problem. *International Journal of Production Research*, 28(10):1817–1830, 1990.

[68] J. P. Stinson and A. W. Smith. A heuristic programming procedure for sequencing the static flowshop. *International Journal of Production Research*, 20(6):753–764, 1982.

[69] W. Szwarc and J. N. D. Gupta. A flow-shop with sequence-dependent additive setup times. *Naval Research Logistics Quarterly*, 34(5):619–627, 1987.

[70] E. Taillard. Some efficient heuristic methods for the flow shop sequencing problem. *European Journal of Operational Research*, 47(1):65–74, 1990.

[71] E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, 1993.

[72] S. Turner and D. Booth. Comparison of heuristics for flow shop sequencing. *OMEGA The International Journal of Management Science*, 15(1):75–85, 1987.

[73] J. A. A. van der Veen and S. Zhang. A linear-time algorithm for sequencing jobs with a fixed number of job-classes. Working Paper TI 94-86, Tinbergen Institute, Erasmus Universtity Rotterdam, The Netherlands, July 1994.

[74] C. H. White and R. C. Wilson. Sequence dependent set-up times and job sequencing. *International Journal of Production Research*, 15(2):191–202, 1977.

[75] M. Widmer and A. Hertz. A new heuristic method for the flow shop sequencing problem. *European Journal of Operational Research*, 41(2):186–193, 1989.

[76] S. Zdrzałka. Preemptive scheduling with release dates, delivery times and sequence independent setup times. *European Journal of Operational Research*, 76(1):60–71, 1994.

Vita

Roger was born in Monterrey, Nuevo León, México on April 16, 1966, the eldest son of Griselda Mercado and Rogelio Ríos. He received a *Licenciado* in Mathematics degree from Universidad Autonóma de Nuevo León in 1988. He earned his M.S.E. and Ph.D. in Operations Research and Industrial Engineering from the University of Texas at Austin in 1992 and 1997, respectively.

Before joining graduate school, he spent four years working at Vitro Tec, a research and development firm belonging to Vitro, Mexico's largest enterprise in the glass industry, where he conducted independent research on mathematical models and algorithms for simulating glass-forming processes. As a graduate student at UT-Austin, he worked as an assistant instructor and as a teaching assistant in various graduate and undergraduate courses. In addition, he also worked as a graduate research assistant. His graduate research has focused on optimization of flow line machine scheduling problems.

His professional duties include membership to Institute for Operations Research and Management Science (INFORMS), Mathematical Programming Society, Society for Industrial and Applied Mathematics, *Sociedad Matemática Mexicana*, American Mathematical Society, and Mathematical Association of America. He has served as the UT-Austin Chapter President of the Omega Rho Honor Society of INFORMS, and as a referee for the Journal of Heuristics. His research work and academic achievements have won numerous awards.

On a more personal note, he has been very happily married to Ofelia Rodríguez since August 1992, and made his debut as a dad on March 14, 1996

when his son, Vandari Pavel, was born.

Permanent address: Caracas 238, Valle del Nogalar
San Nicolás de los Garza, N.L. 66480
México

This dissertation was typeset with LaTeX[‡] by the author.

[‡] LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.