SCHEDULING REENTRANT FLEXIBLE JOB SHOPS WITH SEQUENCE DEPENDENT SETUP TIMES

by

Deepu Philip

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Industrial and Management Engineering

MONTANA STATE UNIVERSITY
Bozeman, Montana

May 2005

APPROVAL

of a thesis submitted by

Deepu Philip

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the college of Graduate Studies.

Dr. Edward L. Mooney

Approved for the Department of Mechanical and Industrial Engineering

Dr. Jay Conant

Approved for the College of Graduate Studies

Dr. Bruce R. McLeod

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available for borrowers under the rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permissions for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Deepu Philip

May 16, 2005

TABLE OF CONTENTS

LIST OF TABLES v

LIST OF FIGURES vii

1. INTRODUCTION 1

Reentrant Flexible Jobshop 2
Related Problems 5

2. SOME USEFUL RFJSSDS PROPERTIES 11

3. A SIMULATION BASED LOCAL SEARCH 27

Local Search Algorithm 28
Candidate Task Identification 30
Filter and Rank Moves 38
Move Filtering Criteria-1: Slack Window 39
Move Filtering Criteria-2: Reentrancy Limit 42
Move Ranking Criteria: A* Move Ordering 42
Simulation Model 45

4. PERFORMANCE EVALUATION 48

RFJSSDS Problem Generator 48
RFJSSDS Results 55
SBLIMS Algorithm Tuning 56
Algorithms for comparison 60
Computational Experimental Design 61
RFJSSDS Results 63
Classical Job Shop Results 77
Reentrant Flow Shop Results 84
Flexible Job Shop Results 86

5. CONCLUSIONS 89

REFERENCES CITED 91
APPENDICES 95
APPENDIX A - PROBLEM GENERATOR 96
APPENDIX B - SIMULATION 115
APPENDIX C - PERFORMANCE EVALUATION 149

LIST OF TABLES

Table		Page
1	Example problem route	32
2	RFJSSDS problem parameter list	57
3	List of compared algorithms on RFJSSDS domain	60
4	Fractional Factorial Experimental Design	64
5	% Gap comparison for problem set 1: N=5, M=2, L=0.3	65
6	% Gap comparison for problem set 1: N=5, M=2, L=0.3	66
7	% Gap comparison for problem set 1: N=5, M=2, L=0.7	67
8	% Gap comparison for problem set 1: N=5, M=2, L=0.7	68
9	% Gap comparison for problem set 2: N=5, M=4, L=0.3	69
10	% Gap comparison for problem set 2: N=5, M=4, L=0.3	70
11	% Gap comparison for problem set 2: N=5, M=4, L=0.7	71
12	% Gap comparison for problem set 2: N=5, M=4, L=0.7	72
13	% Gap comparison for problem set 3: N=10, M=4, L=0.7	73
14	% Gap comparison for problem set 3: N=10, M=4, L=0.7	74
15	Solution time summary for set 1: N=5, M=2, L=0.3/0.7 problems	74
16	Solution time summary for set 2: N=5, M=4, L=0.3/0.7 problems	75
17	Solution time summary for set 3: N=10, M=4, L=0.3/0.7 problems	75
18	Solution Quality of SBLIMS algorithm on ‘abz’ problems	79
19	Solution Quality of SBLIMS algorithm on ‘car’ problems	79
20	Solution Quality of SBLIMS algorithm on ‘la’ problems (5 m/c)	80
21	Solution Quality of SBLIMS algorithm on ‘la’ problems (10 m/c)	81
22	Solution Quality of SBLIMS algorithm on ‘la’ problems (15 m/c)	81
23	Solution Quality of SBLIMS algorithm on ‘mt’ problems	82

24 Solution Quality of SBLIMS algorithm on 'orb' problems 82
25 Comparison of algorithms on 'la' problems (10 m/c) 83
26 Parameter Summary of RFS synthetic problems 85
27 Solution Quality of SBLIMS algorithm on 're305' problems 86
28 Solution Quality of SBLIMS algorithm on 'MT10-FJS' problems . . . 87
29 Solution Quality of SBLIMS algorithm on 'LA24-FJS' problems . . . 87
30 Solution Quality of SBLIMS algorithm on 'LA40-FJS' problems . . . 88
31 List of input parameters for the problem generator 97
32 List of input parameters for the simulator 116

LIST OF FIGURES

Figure		Page
1	Reentrant Flexible Job Shop job routing	3
2	Scheduling Problem Space Diagram	6
3	Deadlock due to reentrancy constraint violation	14
4	Setup for P(1) case of theorems	15
5	Modified schedule for P(1) after performing the left move	16
6	Initial feasible schedule generated for P(r) case	17
7	Modified schedule for P(r) after left move	18
8	Modified schedule for P(1) after right move	19
9	Modified schedule for P(r) after right move	20
10	Comparison of non-delay and active schedule for C_{max}	22
11	A general case schedule for RFJSSDS problem	23
12	Initial solution before performing swap of tasks 20 and 5	24
13	Resultant deadlock after the swap move of tasks 20 and 5	25
14	Local search algorithm overview	27
15	Flow of logic in SBLIMS algorithm	31
16	Random dispatch rule logic	33
17	Non-delay schedule for the illustrative example	34
18	Usable and unusable slack in both task and resource views	36
19	Critical path diagram for the example	38
20	Candidate task list generation algorithm	39
21	Time window for the given example	40
22	Insert of a candidate list task before another task on same machine	41
23	Violation of reentrancy constraint resulting infeasibility	43

24 RFJSSDS problem simulator logic 46
25 Problem Generator Logic 50
26 Route generation procedure 51
27 Processing time sampling procedure 52
28 Sequence dependent setup time sampling procedure 53
29 Arrival times sampling procedure 54
30 Due date sampling procedure 55
31 Algorithm rankings for RFJSSDS problems 76

ABSTRACT

This study presents a new simulation-based local search approach for solving shop scheduling problems. Results for classical problems from the literature demonstrate the effectiveness and quality of the approach. Application is also shown for reentrant flexible job shop with sequence dependent setup times (RFJSSDS), a new, very general, class of problems. RFJSSDS is a generalization of the classical job shop, reentrant flow shop and flexible job shop problems. Multiple products (routes), sequence dependent setup times at the work centers and reentrancy of the jobs make RFJSSDS one of the more general and difficult shop scheduling problems. Examples of this type of problem include semiconductor wafer fabrication facilities and flexible machining systems. The solution methodology developed in this study features a new Simulation Based Local Improvement with Multi Start (SBLIMS) algorithm. The local search procedure modifies an initial feasible solution provided by the simulation module to generate promising neighbor solutions. A generated solution is considered to be better if there is a reduction in the total completion time or makespan. A unique filtering strategy is used to select and rank moves, using both task and resource views of a schedule. Multiple random starting points are generated in multistart fashion as part of the solution process. New theorems are presented that form the basis for SBLIMS. The SBLIMS algorithm was evaluated using test instances for several shop scheduling problems as well as RFJSSDS. A set of synthetic problems was generated to study RFJSSDS, because there were no RFJSSDS instances available from the literature. The SBLIMS algorithm was compared with various dispatch rules in the RFJSSDS domain and its performance was found to be better in most cases. SBLIMS was also tested with well known special cases of RFJSSDS: the classical job shop, reentrant flow shop and flexible job shop problems. The SBLIMS algorithm provided excellent results compared with those provided in the literature, establishing the generality of the approach for solving a broad class of shop scheduling problems.

CHAPTER 1

INTRODUCTION

Scheduling is the process of allocating limited resources to tasks over time to optimize one or more objectives. Resources might be machines or people in a shop, material handling systems, etc. Examples of tasks include basic machining operations, moving, transporting, loading, unloading, part programming etc. Tasks may have earliest start times and due dates and some tasks might have priority over others. Scheduling objectives include minimizing the completion time (makespan) for a set of tasks, minimizing lateness, maximizing the number of tasks completed in a given time, minimizing work in process inventory, etc.

Shop scheduling problems are distinguished by linear task precedences. The reentrant Flexible Job Shop with Sequence Dependent Setups(RFJSSDS) problem is a very general shop scheduling problem. RFJSSDS allows general job routing between work centers with parallel resources at each work center. Semiconductor manufacturing, which is characterized by astronomical capital costs, short product life, dynamic demands etc.[1], can be modelled using RFJSSDS.

This research focused on developing an effective solution algorithm for solving RFJSSDS. Since not much work has been done on RFJSSDS, test problems were not available. However, results were available for restrictions of RFJSSDS: Reentrant Flow Shops (RFS), Flexible Job Shops (FJS), Flexible Flow Lines(FLL) problems. Thus, this thesis presents the following research results:

- A new RFJSSDS problem generator and set of synthetic test problems for RFJSSDS for research.
- Insights into the RFJSSDS domain that will help understanding the problem better.
- A general purpose algorithm for solving RFJSSDS and its restrictions.
- Computational results documenting the performance of the algorithm and its generality.

Reentrant Flexible Jobshop

A schematic diagram of the flow of two jobs through a reentrant flexible job shop is given in Figure 1 to illustrate reentrancy. Jobs may be routed through w work centers (fabs [1], stages [2], machine centers [3], facilities [4]) in a predetermined sequence. Each work center has at most m parallel, identical resources (units [2], identical resources [3], cell resources [5], machine units [6], [7]). This means that an operation, or task, can be accomplished with any of the resources in a work center.

Jobs move between work centers as through a job shop and might skip certain work centers. Each job can have its own route. Once a job completes one pass through the shop, it may reenter the facility for a different set of operations. The resources at each work center are capable of performing multiple operations so need not be doing the same operation at the work center as on the last pass. Processing times for each reentrant loop are specified by the route for the particular job, which is assumed to

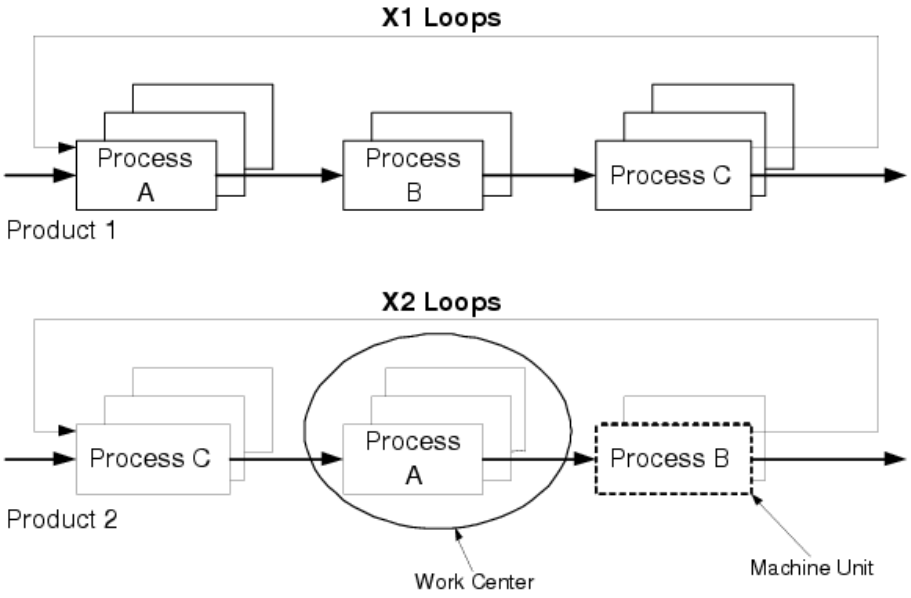

Figure 1: Reentrant Flexible Job Shop job routing

be known. At every work center jobs incur sequence dependent setup times along with the processing time. Processing and sequence dependent setup times are also assumed to be known.

The objective assumed for RFJSSDS in this study is to minimize the maximum completion time (makespan), denoted by C_{max}. The RFJSSDS problem can formulated using the disjunctive programming approach. Let

$R = \{r : r = 1, 2, ..., n_r\}$, the resource indices set.

$W = \{w : w = 1, 2, ..., n_w\}$, the set of work centers.

$R_w \subseteq R$, a subset of resources in work center w, partitioning R into n_w subsets.

$I = \{i : i = 0, 1, 2, ..., n_i, n_{i+1}\}$, the set of tasks

where

n_i = number of original tasks and

task 0 is the common start task, task n_{i+1} is the common finish task.

$w(i)$ = the known work center for task i = 1,2,..,n_i.

$$x_{ijr} = \begin{cases} 1 \text{ if task } i \text{ immediately precedes task } j \text{ on resource } r \in R_{w(i)}, R_{w(j)} \\ 0 \text{ otherwise} \end{cases}$$

t_i = start time for task i.

C_i = completion time for task i.

$= t_i$ + processing time + setup time ($C_0 = t_0$, $C_{n_i+1} = t_{n_i+1}$.

p_i = processing time for task i.

$$s_{ij} = \begin{cases} \text{sequence dependent setup time for task } j \\ \text{when preceded by } i \text{ where } r \in R_{w(i)} = R_{w(j)}. \\ 0 \text{ otherwise} \end{cases}$$

$P = \{(i,j) : \text{task } j \text{ follows } i, \text{ where } (i,j) \text{ belong to the same job}\}$.

Now we want to find x_{ijr} and t_i for all i, j and r to minimize the maximum completion time or makespan. That is we want to

$$\text{Minimize } \max_i(C_i) \tag{1.1}$$

subject to:

$$C_i = t_i + p_i + \sum_i \sum_{j:w(j)=w(i)} \sum_{r \in R_{w(i)}=R_{w(j)}} x_{ijr} s_{ij} \text{ for all } i \tag{1.2}$$

$$t_j \geq C_i \text{ for all} (i,j) \in P \tag{1.3}$$

$$\sum_{j \neq i, w(j)=w(i)} x_{ijr} = 1 \text{ for all } i > 0, r \in R_{w(i)} \tag{1.4}$$

$$\sum_{i \neq j, w(i)=w(j)} x_{ijr} = 1 \text{ for all } j \leq n_i, r \in R_{w(j)} \tag{1.5}$$

and

$$t_i \geq 0 \text{ for all } i; \quad x_{ijr} \in \{0,1\} \text{ for all i, j, r.}$$

Equation 1.1 is the objective function of RFJSSDS problem formulation. Equation 1.2 explains how the completion time is calculated from start time of any task i. Equation 1.3 enforces job-based task precedences. Equations 1.4 and 1.5 represent the resource conflict, or disjunctive constraints. Equation 1.4 states that each task has exactly 1 immediate successor on resource r and Equation 1.5 states that each task has exactly 1 immediate predecessor on resource r, except the start and end tasks.

Related Problems

RFJSSDS is a generalization of the Flexible Job Shop problem (FJS) where routes can be repeated and setups are sequence dependent. Likewise FJS is a generalization of the classical job shop scheduling problem [1], [2], [6], [8], [9]. Shop scheduling problems can be represented on a three dimensional cube with the X, Y and Z axis representing Resource types, Precedences and Resource capacity respectively as shown

in Figure 2, developed by Mooney [10]. RFJSSDS is a generalization of Flexible Job Shop (FJS) as shown in Figure 2.

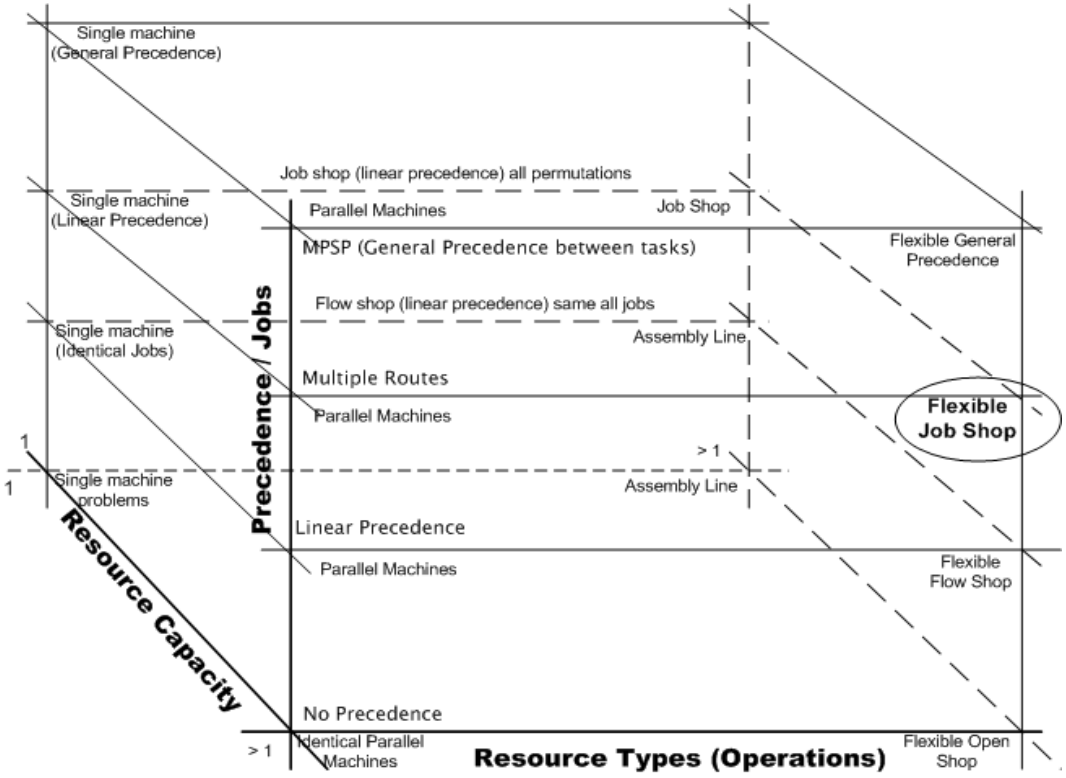

Figure 2: Scheduling Problem Space Diagram

The single machine scheduling problem with sequence dependent setup times is equivalent to the Traveling Salesman Problem (TSP) [11] and is NP-Hard. This means FJS with sequence dependent setup times is NP hard, because single machine scheduling is a restriction of FJS (or FJS is a generalization of single machine scheduling problem). Similarly, since RFJSSDS is a generalization of FJS, it is also NP-Hard and exact solution is computationally impossible [12] for large instances.

Reentrancy was described by Graves et.al. [4] during a study of semiconductor fabrication process. Uzsoy, Lee and Martin-Vega [9] presented an excellent description of problems associated with production planning in semiconductor industry. Work on different solution techniques is summarized as well in the paper [9]. Jeong and Lee [13] assessed four types of lot release rules and three dispatch rules in different semiconductor manufacturing environments. The authors made a best and worst choice among these rules. The study was done using simulation models with the comparison based on the mean throughput rate and throughput times for each rule. They concluded that the lot release rules performed marginally better than the dispatch rules.

A number of authors have studied various restrictions of the reentrant flexible job shop problem. Demirkol and Uzsoy [14] presented a set of decomposition methods to solve the reentrant flow shop RFS problem, a restriction of RFJS. They used rolling horizon heuristics presented by Ovacik and Uzsoy [15] to decompose the problem and later enhanced them with tabu search. Mastrolilli and Gambardella [8] presented a local search method with two neighborhood functions for RFS. The main contribution was the reduction of the set of neighborhoods to a subset, which can be proved to contain the neighbor with lowest makespan always.

Liu and Chang [16] presented a Lagrangian relaxation approach to schedule a flexible flow shop. A mathematical formulation of the flexible flow shop problem (FFS) was presented in this paper. Yang, Kreipl and Pinedo [2] presented a set of

three heuristics to solve the flexible flow shop problem and provided evidence that their hybrid algorithm combining local search and decomposition methods yielded better results. Ding and Kittichartphayak [17] presented three different heuristics to schedule flexible flow lines. FFL is very similar to FFS, except that the goal is the optimization of product flow as in the case of an assembly line. Leon and Ramamoorthy [18] proposed a problem space based search method for flexible flow line scheduling. Near optimal solutions and significant improvements are shown by the authors using single pass heuristics. Chambers and Barnes [6] have proposed a reactive search to solve flexible job shop scheduling problems. The results are compared to the dynamic, adaptive tabu search by the same authors to provide enough proof for the superiority of the reactive tabu search.

There are some other variations of the RFJSSDS problem treated extensively in the literature. Nowicki and Smutnicki [3] demonstrated a tabu search approach on a parallel machine flow shop. Negenman [19] gave some local search algorithms for parallel machines problem. Brah and Loo [20] worked on heuristics for the multiprocessor flow shop problem. Daniels, Hua and Webster [21] addressed the parallel machine flexible resource-scheduling problem with unspecified job assignments. An extension of the shifting bottleneck approach to the parallel machine flowshop scheduling problem was presented by Sung, Kim and Yoon [22].

The shifting bottleneck (SB) heuristic has been adapted by Mason et.al. [23] to schedule complex job shop environment. The complex jobs shop is a reentrant

flexible job shop (RFJS) with limited products (routes) and unreliable resources. This formulation extends the typical disjunctive graph formulation to accommodate batch processing steps, reentrant product flow and sequence dependent setups on certain resources. The objective is to minimize the total weighted tardiness. Various rescheduling strategies to minimize the total weighted tardiness in complex job shops are explained in Mason et.al. [24]. The methods of right shift rescheduling, fixed sequence rescheduling and total rescheduling are compared in this study. The emphasis of the study is on how to deal with machine breakdowns in complex job shops using various rescheduling strategies.

Other studies have used elements of our solution approach. This thesis was motivated by the previous work done by Mooney and Kerbel [1] on multi product flow shop with reentrancy and sequence dependent setups. The study was conducted using a simulation tool to assess the cost and benefits of introducing a second product into the shop floor. A Sequential task elimination method was used by Toth and Vigo [25]. These authors named it a granular approach and was used it to solve vehicle routing problems. The ranking and ordering of the moves in this research were based on the A* algorithm described in Russell and Norvig [12].

This chapter introduced the RFJSSDS problem and its relationship to other shop scheduling problems. The remainder of the thesis is organized as follows. Chapter 2 provides a framework for the solution algorithm. Details of the local search algorithm developed for RFJSSDS are discussed Chapter 3, followed by performance evalua-

tion and findings in Chapter 4. Chapter 4 also provides RFJSSDS synthetic problem generation details. Chapter 5 summarizes the study and its main conclusions. Appendices provide analysis and results details.

CHAPTER 2

SOME USEFUL RFJSSDS PROPERTIES

During this study of the RFJSSDS problem, some properties were identified that revealed important insights for developing the simulation based shop scheduling algorithm. Since these properties have not been addressed previously in the scheduling literature, we present them in this chapter as RFJSSDS theorems with proof. Necessary assumptions are:

- Jobs are sets of ordered tasks.
- The task precedences follow a classical shop precedence pattern where each task has atmost one successor, predecessor, or both.
- All time values are deterministic.
- At least one job will reenter the shop one or more times.

The notations used for the theorems are as follows.

$m =$ loop number with values from {1,2,3,..,n_x}.

$n =$ operation number with values from {1,2,3,..,n_r}.

$u =$ job number with values between {a,b,c,..,z}.

$i =$ task index with values {1,2,3,..,n_q}.

$j =$ work center index with values from {1,2,3,..,n_y}.

$g^u_{mn} =$ a task of m^{th} loop, n^{th} operation and u^{th} job.

WC-j = the work center upon which the tasks of a job are processed.

$t^u_{mn} =$ start time of the task g^u_{mn}.

$p^u_{mn} =$ processing time (including setup time) of the task g^u_{mn}.

$f^u_{mn} =$ the finish time of the task g^u_{mn}, which is equivalent to $t^u_{mn} + r^u_{mn}$.

In principle a job shop problem can have an infinite number of feasible schedules. This is possible because an arbitrary amount of idle time can be inserted at any resource between the adjacent pair of tasks, as shown by Baker [26]. In a RFJSSDS problem, we cannot have infinite number of feasible schedules because some schedules will result in deadlock by violating the reentrancy constraints as shown by the following theorem.

Theorem 1 (Deadlock). *In an RFJSSDS problem, any schedule violating the constraint* $t^u_{(m+1)n} \geq t^u_{mn} + r^u_{mn}$ *on a work center, where* $t^u_{(m+1)n}$ *is the start time of the* n^{th} *operation of* $(m+1)^{st}$ *loop, will result in a deadlock.*

Proof :

To prove this theorem we extend the job shop theorem by Baker [26].

For the rest of this thesis, we call the constraint $t^u_{(m+1)n} \geq t^u_{mn} + r^u_{mn}$,

the *reentrancy constraint.* From Graves et.al [4], the schedule within a reentrant loop is equivalent to a job shop, for which the theorem of Baker, stated in the previous paragraph, holds. Now consider a schedule that violates the reentrancy constraint on the same work center, which results in $t^u_{(m+1)n} < t^u_{mn} + r^u_{mn}$. By the job based precedence constraint given in Equation 1.3, the task denoted by g^u_{mn} should occur before task $g^u_{(m+1)n}$. But the schedule has the start time of task $g^u_{(m+1)n}$ set before the start time of task g^u_{mn}. This results in a situation where task $g^u_{(m+1)n}$ will wait for the occurrence of task g^u_{mn}, which will never happen because it is scheduled later on the same work center. Any amount of idle time between the tasks will not resolve this because the tasks are on the same work center, but belong to different reentrant loops. This results in a deadlock that cannot be resolved; hence it is an infeasible schedule. Such an infeasible schedule is shown in Figure 3 for job p on work center 2.

Thus reentrancy requires that a task not be scheduled before its job's previous loop's task, on the same work center, implying that the operations in successive loops cannot be started until all the operations in predecessor loops are finished. We use the reentrancy constraint to prove the rest of the theorems.

To prove the remaining theorems in this chapter, we use mathematical induction. For all theorems, the initial case of the proof, denoted by P(1), is a problem with 2 jobs each with 4 tasks to be processed on 2 work centers in 2 loops. The general case,

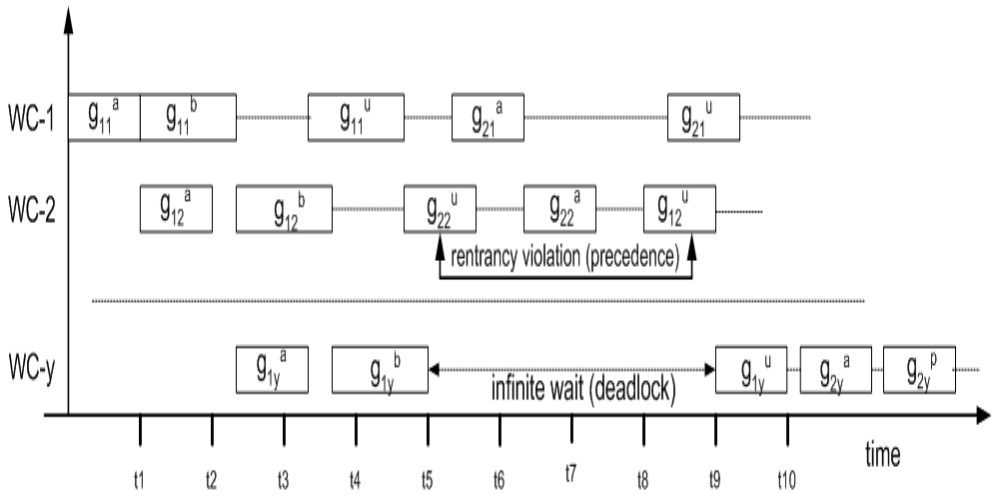

Figure 3: Deadlock due to reentrancy constraint violation

denoted by P(r), has z jobs, each with q tasks to be processed on y work centers in x loops. The final case, denoted by P(r+1), has $(z+1)$ jobs each with $(q+1)$ tasks to be processed on $(y+1)$ work centers in $(x+1)$ loops.

The P(1) case of all the theorems in this chapter uses the initial feasible schedule represented in Figure 4, as a Gantt chart. We have 2 work centers denoted by WC-1 and WC-2, two jobs denoted by a and b, with tasks labelled according to the notational conventions described at the beginning of this chapter.

Theorem 2 (Left move). *In a RFJSSDS problem schedule, a left move of a task on a work center without violating the reentrancy constraint will result in a feasible schedule.*

Proof :

P(1) : The sequence on work center (WC-1) is g^a_{11} - g^b_{11} - g^a_{21} - g^b_{21} with

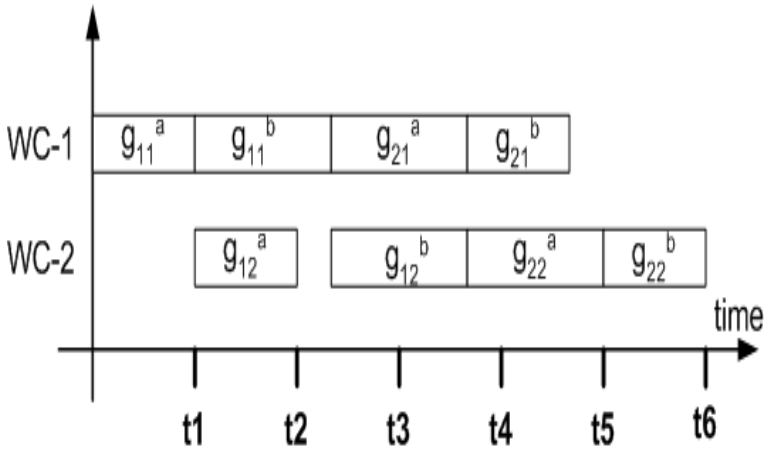

Figure 4: Setup for P(1) case of theorems

the corresponding start times t^a_{11}, t^b_{11}, t^a_{21}, t^b_{21} and the corresponding finish times f^a_{11}, f^b_{11}, f^a_{21}, f^b_{21}. Left move of the task in the 3^{rd} sequence position, g^a_{21} to 2^{nd} sequence position on WC-1 results in the modified sequence given by g^a_{11} - g^a_{21} - g^b_{11} - g^b_{21}. Since $t^a_{21} \geq f^a_{11}$ the reentrancy constraint is satisfied. The modified schedule is given in Figure 5, which is still feasible. This proves the case of P(1).

P(r) : Consider the general case where there are q jobs, having n tasks to be processed on y work centers in x loops. Figure 6 shows the general case. The sequence on work center (WC-1) is g^a_{11} - g^b_{11} - g^p_{11} - g^a_{21} - g^p_{21}. Let the sequence position of g^p_{21} be r, sequence position of g^p_{11} be h.

The left move and insert operation without violating the reentrancy constraint will put g^p_{21} at any position k, such that $t^p_{21} \geq f^p_{11}$ and $h < k < r$. This gives the new sequence g^a_{11} - g^b_{11} - g^p_{11} - g^p_{21} - g^a_{21}. The schedule is

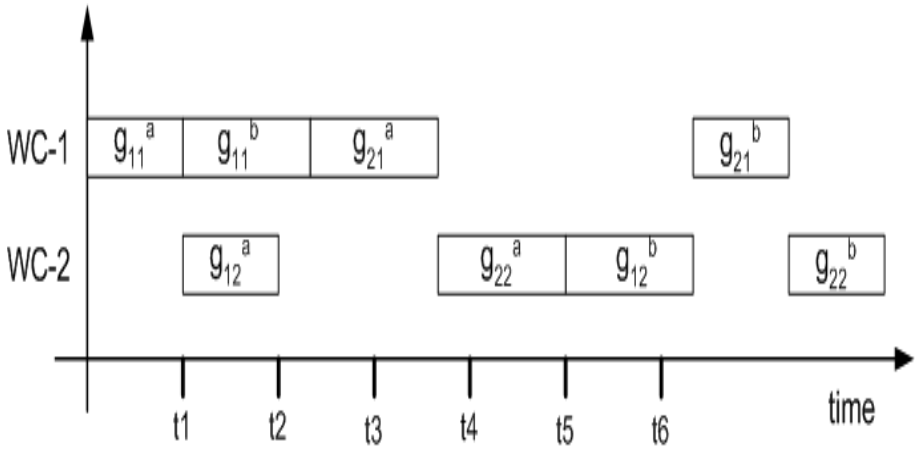

Figure 5: Modified schedule for P(1) after performing the left move shown in Gantt chart in Figure 7.

The new schedule shows that the left shift without violating the reentrancy constraint guarantees a feasible schedule again. This is due to the fact that any amount of idle time can be inserted between two adjacent tasks on the same work center, which helps to maintain the feasibility of the schedule, which proves P(r). Now consider the P(r+1) case, where the task belongs to the $2+1=3$ loop. If we choose tasks g_{31}^{p} and g_{21}^{p}, the same conclusion can be reached on a left move without violating the reentrancy constraint. Since we already proved that P(1) and P(r) are true, and P(r+1) is also true for any P(r), which proves the Theorem 2.

Theorem 3 (Right move). *In a RFJSSDS problem schedule, a right move of a task on a work center without violating the reentrancy constraint will result in a feasible schedule.*

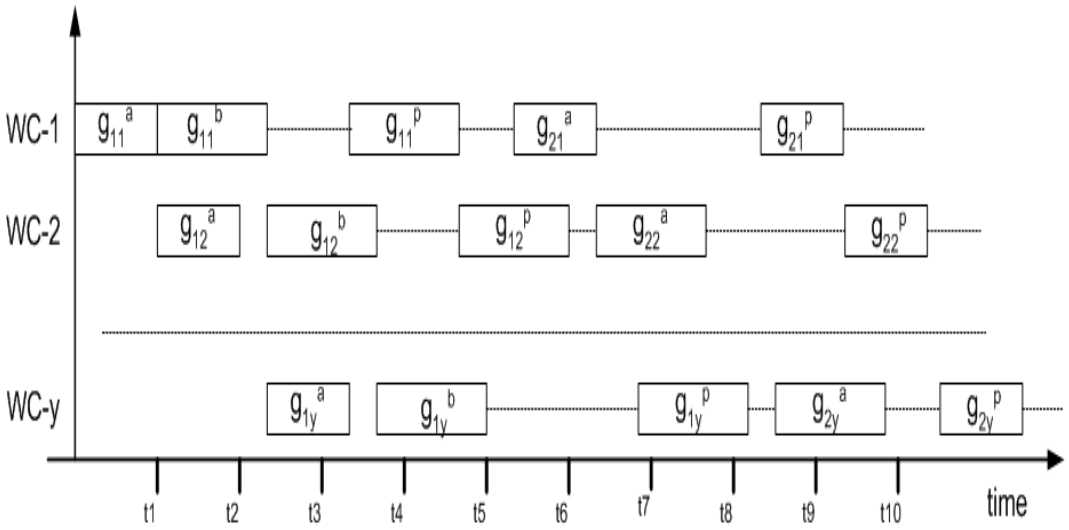

Figure 6: Initial feasible schedule generated for P(r) case

Proof :

P(1) : Consider the schedule represented in Figure 4. The sequence on work center (WC-2) is given by the order g^a_{12} - g^b_{12} - g^a_{22} - g^b_{22}, with the start times given by t^a_{12}, t^b_{12}, t^a_{22}, t^b_{22} as well as the finish times for the tasks in the sequence is given by f^a_{12}, f^b_{12}, f^a_{22}, f^b_{22}. Performing a right move on task in sequence position two (g^b_{12}) to sequence position three on work center (WC-2) results in the modified sequence g^a_{12} - g^a_{22} - g^b_{12} - g^b_{22}. Since $t^b_{22} >= f^b_{12}$, the reentrancy constraint holds good. The modified schedule is shown in Figure 8. The schedule is still maintains its feasibility, proving the P(1) case of Theorem 3.

P(r) : Consider a more general case as shown in Figure 6. The sequence on work center (WC-2) is given by the order g^a_{12} - g^b_{12} - g^p_{12} - g^a_{22} - g^p_{22}. Let

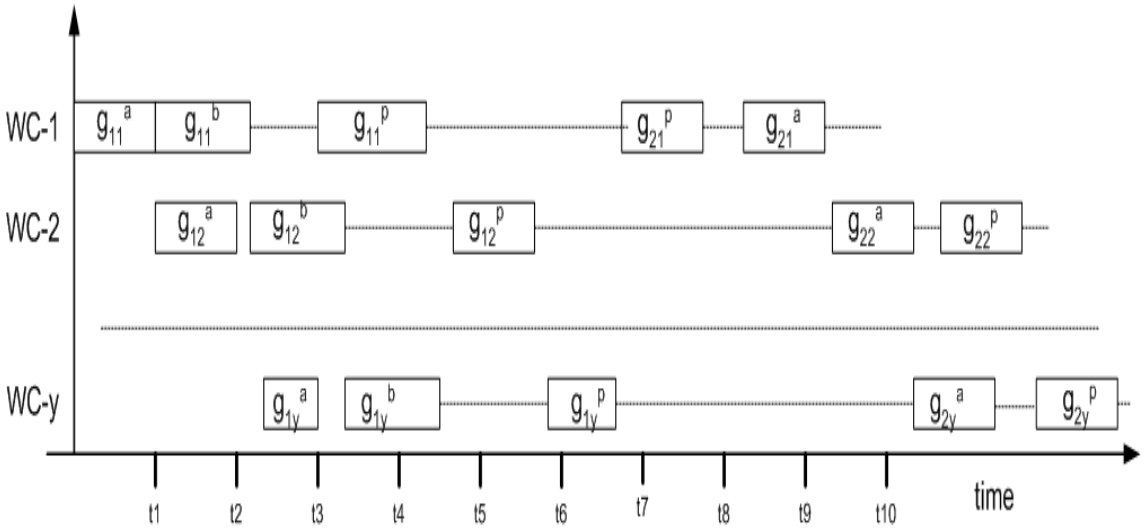

Figure 7: Modified schedule for P(r) after left move

the sequence position of task g_{12}^p is c and g_{22}^p is s. The right move and insert operation without violating the reentrancy constraint will put g_{12}^p at any position k, such that $t_{22}^p \geq f_{12}^p$ and $c < k < s$. So the modified schedule on work center (WC-2) can be stated as g_{12}^a - g_{12}^b - g_{22}^a - g_{12}^p - g_{22}^p. The modified schedule represented as Gantt chart is shown in Figure 9.

The new schedule shows that any right move and insert operation without violating the reentrancy constraint guarantees a feasible schedule. The arbitrary amount of idle time that can be inserted between any two adjacent tasks helps to maintain feasibility. This proves the P(r) case of the theorem. Now consider the case where task g_{32}^p and g_{22}^p, where the loop is $2 + 1 = 3$, denoting the P(r+1) case. Using the same argument for the P(r) proof, all schedules are feasible if $t_{32}^p \geq f_{22}^p$ (reentrancy constraint).

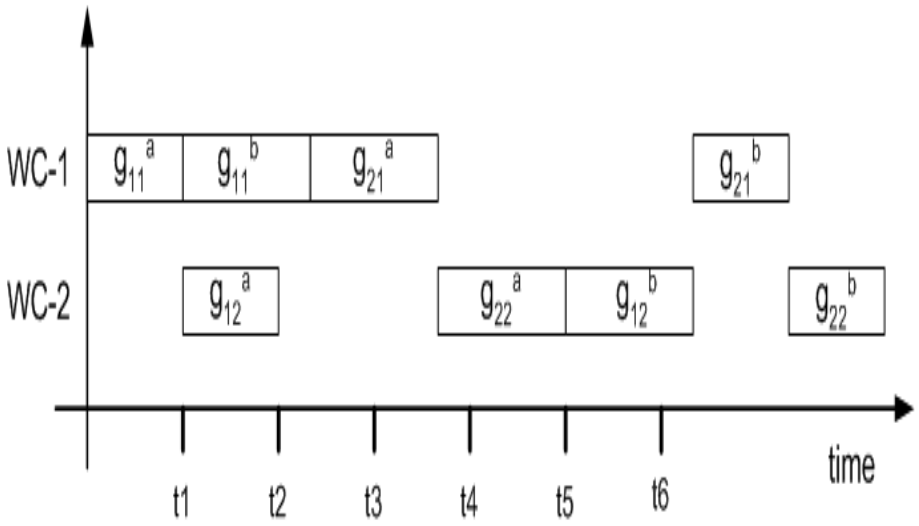

Figure 8: Modified schedule for P(1) after right move

Since the P(1) and P(r) cases of the theorem are proved true and P(r+1) holds good for cases when P(r) is true, which proves Theorem 3.

Lemma 1: *The set of feasible schedules for a reentrant flexible job shop problem is finite.*

Proof : It is quite evident from the Theorems 1, 2 and 3 that in a RFJSSDS problem all permutations do not guarantee a feasible schedule. The set of feasible schedules is a subset of the set of all possible permutations. For a classical job shop problem the set of feasible schedules is infinite in size according to Baker [26], as any permutation guarantees a schedule. Since this is not true for RFJSSDS, we have a finite number of feasible sequences.

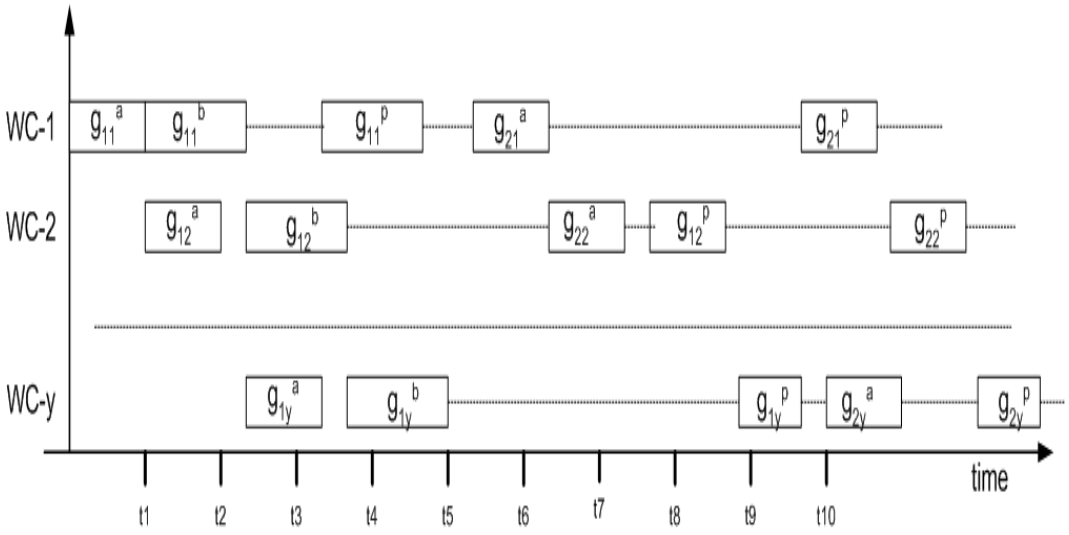

Figure 9: Modified schedule for P(r) after right move

Some observations and definitions from the classical job shop problem can be extended to the RFJSSDS problem too. They are the definitions of semi-active, active and non-delay schedules. These definitions are modified to accommodate for the reentrancy characteristic of RFJSSDS.

Semi-active schedule: Given a feasible sequence for each work center in RFJSSDS, there is only one schedule in which no local left shift (on any work center) can be made. The set of all such schedules is the set of semi-active schedules. The set of semi-active schedules is finite because the set of feasible schedules are finite and any subset of a finite set is also finite.

Active schedule: Given a feasible schedule for each work center in RFJSSDS, the set of active schedules forms a subset of semi-active schedules, where no global left shifts are possible (on any of the work centers). The set of active schedules is

finite. To optimize the solution for a RFJSSDS, only the set of active schedules need to be considered. Baker [26] has presented algorithm for generating random active schedules in the case of classical job shop problems.

Non-delay schedule: Given a feasible schedule for each work center in RFJSSDS, a non-delay schedule is a schedule where no machine in a work center is kept idle when there is an operation available for processing. No task is permitted to wait in the queue of the work center when there is an unit of resource available for processing the task.

The set of non-delay schedules is also finite, and it is a subset of active schedules. But there is no guarantee that the set of non-delay schedules will contain the optimal solution. This is shown in the Figure 10, as an example where the active schedule results in a better value of maximum completion time (C_{max}). This example is a situation where a better solution is provided by the active schedule than the non-delay schedule. So to obtain the optimal schedule for any performance measure it is only necessary to search the set of active schedules. This result is a direct extension of the job shop result shown by Baker [26].

Theorem 4 (Swap feasibility). *In a RFJSSDS schedule using simulation, any swap move where the predecessor of the task moving forward in the sequence position has a start time greater than or equal to the end time of the successor of that task that is moving backward in sequence position on the same machine other than where the swap is performed will result in a deadlock.*

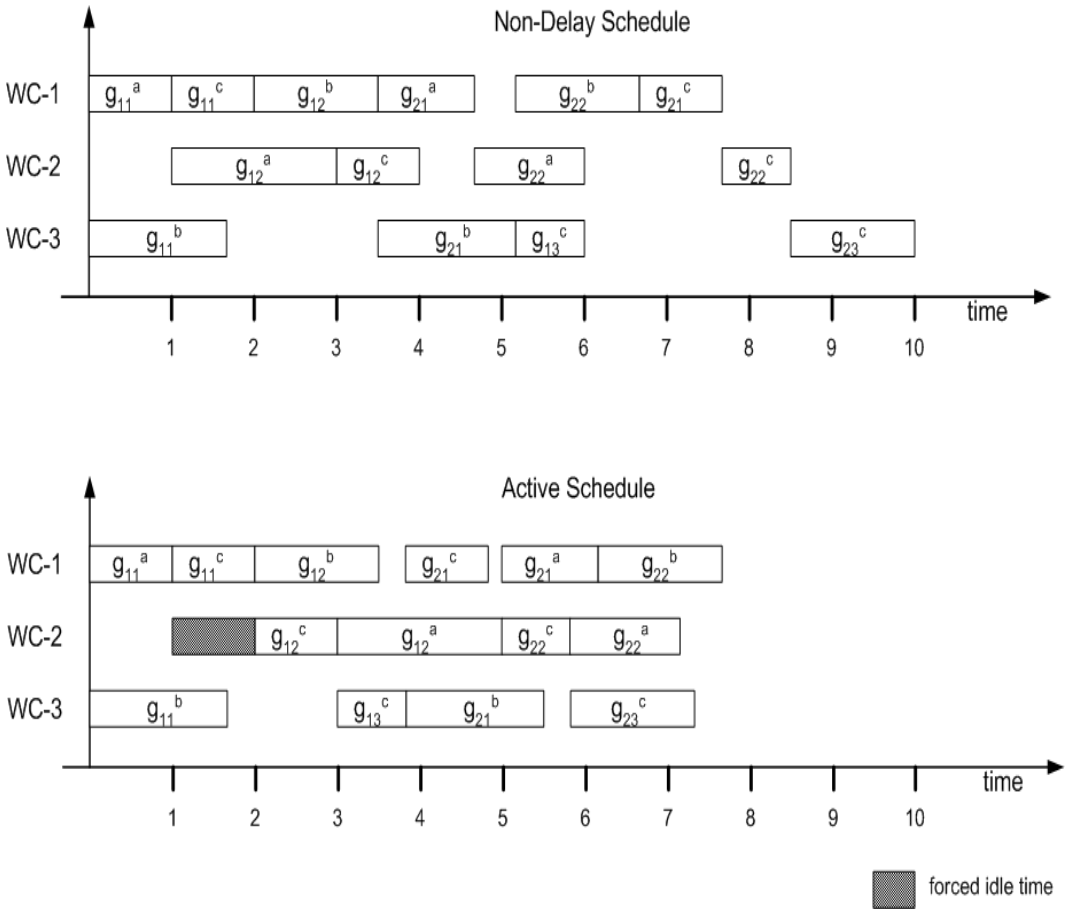

Figure 10: Comparison of non-delay and active schedule for C_{max}

Proof : We prove this theorem by showing a general case scenario. Consider a task on any work center g^u_{mn}, where the task belongs to the m^{th} loop, n^{th} operation and for the u^{th} job. The immediate predecessor of the task is $g^u_{m(n-1)}$ and the immediate successor is $g^u_{m(n+1)}$. Figure 11 shows a schedule for three work centers. Now, assume that the tasks $g^b_{r(s+1)}$ and $g^a_{(i+1)(j+1)}$ are swapped. The swaps do not violate any reentrancy constraints. Hence it is reentrancy feasible. But when the swap

is accomplished, a deadlock results. The deadlock is due to the following reasons:

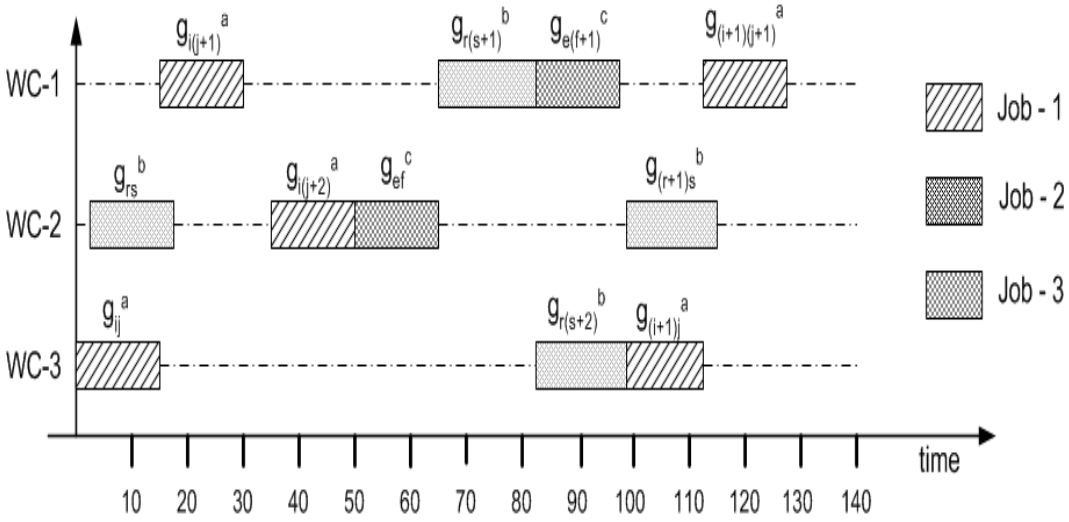

Figure 11: A general case schedule for RFJSSDS problem

- The task $g^a_{(i+1)(j+1)}$, waits for the occurrence of the predecessor, which is $g^a_{(i+1)j}$ on work center 3.

- Task $g^a_{(i+1)j}$ waits for task $g^b_{r(s+2)}$, which is at a lower sequence position on work center 3.

- Task $g^b_{r(s+2)}$ waits for the completion of its predecessor $g^b_{r(s+1)}$ on work center 1.

- Task $g^b_{r(s+1)}$ waits for $g^a_{(i+1)(j+1)}$ on work center 1, because after the swap $g^b_{r(s+1)}$ is at a higher sequence position on the work center.

- Task $g^b_{r(s+2)}$ cannot begin because its predecessor task has not completed and

$g^a_{(i+1)(j+1)}$ has to finish for $g^b_{r(s+1)}$ to start. But $g^b_{r(s+1)}$ cannot start because the predecessor of $g^a_{(i+1)(j+1)}$, $g^a_{(i+1)j}$ is blocked by $g^b_{r(s+2)}$, resulting in the deadlock.

Theorem 4 can be better demonstrated by the following example. Figure 12 shows the initial feasible schedule upon which a swap move was executed. The tasks selected for swap moves were 5 and 20 at sequence position four and six at the work center (WC-1).

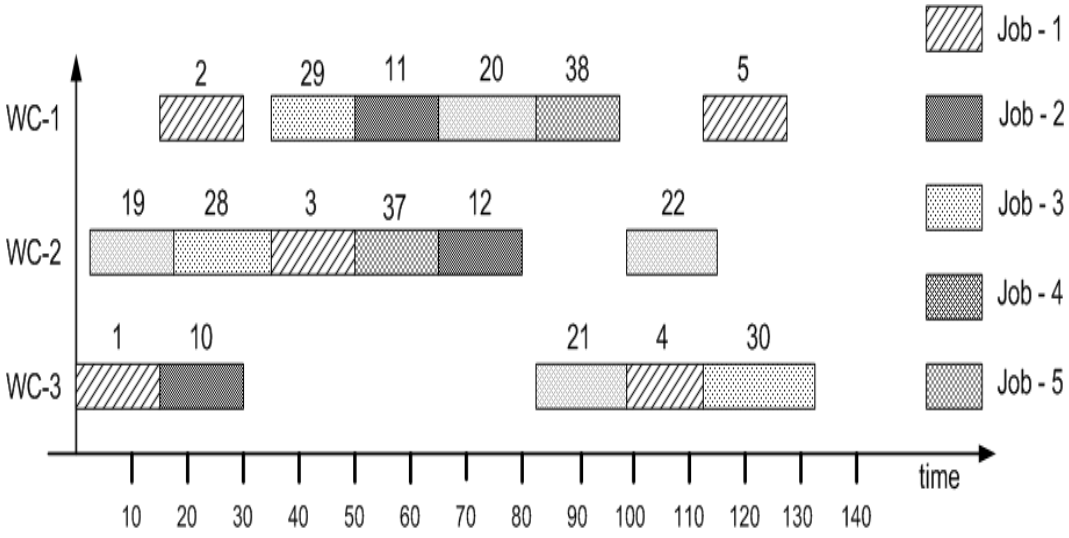

Figure 12: Initial solution before performing swap of tasks 20 and 5

A swap between 5 and 20 does not violate any of the reentrancy constraints. But on close examination of work center (WC-3), task 21, the successor of task 20, precedes task 4 in the sequence. Task 4 is the predecessor of task 5. Upon swapping tasks 20 and 5, the following results:

- Task 5 waits for its predecessor task 4 to complete.
- Task 4 waits for task 21 to compete because of the given sequence order.
- Task 21 waits for its predecessor task 20 to be complete.
- Task 20 waits for task 5 to complete, because after the swap task 5 comes before task 20 in sequence position.

The deadlock is explained with the help of Figure 13. It can also be seen that this will hold for the current example whenever task 4 has a start time greater than or equal to the end time of task 21.

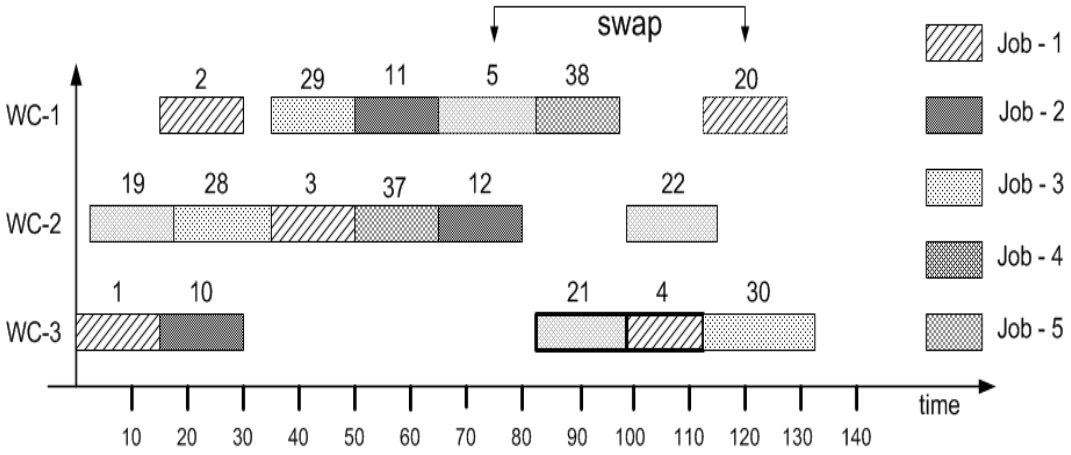

Figure 13: Resultant deadlock after the swap move of tasks 20 and 5

Intuitively this holds for any swap where the predecessor of the task moving ahead (lower sequence position or towards left) has a start time greater than or equal to the end time of successor of task moving backward (or to a higher sequence position or towards right), where they occur on the same work center other than the one upon

which the swap is performed, results in a dead lock.

The next chapter discusses the simulation based local search algorithm (SBLIMS). The theorems and lemmas from this chapter were used in developing the local search algorithm and will be referenced as needed.

CHAPTER 3

A SIMULATION BASED LOCAL SEARCH

The principles developed in Chapter 2 are applied in this chapter to develop a local search algorithm for solving RFJSSDS problems. Specifically, we present a Simulation Based Local Improvement with Multi Start (SBLIMS) algorithm. While SBLIMS was developed for the RFJSSDS problem, we demonstrate its efficacy for a broad class of shop scheduling problems as well.

The generality of the algorithm arises from the fact that it is run over a next event simulation. Since the algorithm uses a simulation model for its move selection and evaluation and no RFJSSDS specific structures are used it could be used to solve other shop scheduling problems.

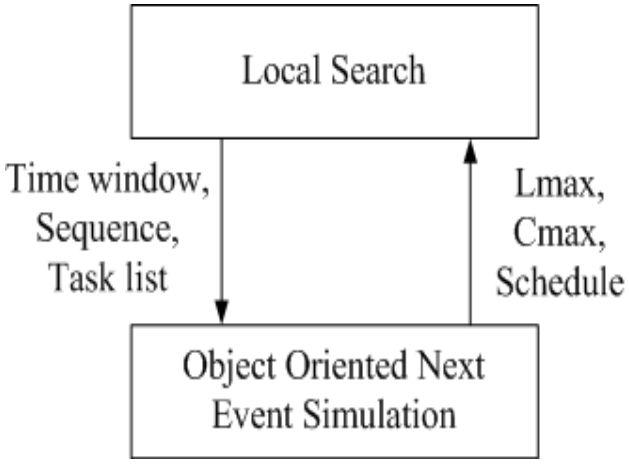

Figure 14: Local search algorithm overview

The relationship between the local search and simulation modules is shown in Figure 14. As shown, the local search module provides the simulation module infor-

mation about the job sequence on each work center, the objective to evaluate, and, when only a part of the schedule is evaluated, the time window parameters. The simulation module provides feedback to the local search module that is used to make decisions. The search terminates when a specified number of iterations are completed without any improvement or when the maximum number of iterations is reached. The final solution is then converted to a schedule and output in both task and resource views.

Since simulation takes a long time to evaluate moves, we incorporated unique features in both the local search algorithm and the simulation model. Techniques are introduced to reduce the number of moves to be evaluated as part of the local search. The simulation works in two modes so that it can be used in various capacities by the local search part of the algorithm. The modes are specified by the local search based on what needs to be done. The modes applicable for the simulation are:

- Mode-1: The simulation works as the initial solution generator and generates a random non-delay schedule for the given instance of RFJSSDS.
- Mode-2: The simulation works as the evaluation tool for any given sequence by the local search and may evaluate only a part of the schedule when possible.

The following section explains the modifications to the local search algorithm to make it more efficient.

Local Search Algorithm

Local search works by perturbing the schedule at each iteration. Changes are made by selecting moves which define a neighborhood about the current solution. The neighborhood is important. For example, a single machine problem with n jobs has $n-1$ neighbors, if the local search performs adjacent pairwise swaps [11]. In the typical job shop problem defined by $J//C_{max}$ notation, a simple neighborhood of semiactive schedules can be obtained by swapping the adjacent tasks on the critical path of the existing schedule [11]. It has been shown by various experiments that the above mentioned neighborhood is not very effective in the case of $J//C_{max}$ problem. In the study of Reentrant Flow Shop problem by Demirkol and Uzsoy [14], the neighborhood is defined by swaps of adjacent tasks on the critical path of the subproblem.

The neighborhood for the SBLIMS algorithm is defined as the left or right move of the best task from the candidate task list to an adjacent position on the same work center. The reasons for choosing the adjacent tasks on the units of work centers are as follows:

- From Theorem 3 we know that any moves violating the reentrancy constraint will result in an infeasible schedule. So by the structure of the problem, there exists a restriction on how we can generate moves.

- From Theorem 4 we know that violating the precedences in the same reentrancy loop will result in an infeasible schedule. This further restricts the moves that need to be considered.

- It has been mentioned before that the RFJSSDS problem has more interactions

between tasks than other shop scheduling problems. So as the distance between the tasks in the sequence position increases, the amount of idle time needed to maintain the feasibility of the schedule increases rapidly.

- The initial solution generated by the simulation lets the task choose the unit of work center with minimum sequence dependent setup times. So any switching of the task to other unit of the work center will add the extra sequence dependent setup times. This will be detrimental in the problem instances where sequence dependent setup times form a sizeable percentage of the production time.

As simulation is used to evaluate moves, we can only evaluate a few moves at each local search iteration due to time constraints. There are three major steps used in the local search to find the best set of moves.

1. Identify a candidate *task* list to generate initial candidate *move* list.
2. Filter and rank moves
3. Evaluate the moves and select the best move

The general logic of the SBLIMS algorithm is given in Figure 15. The algorithm works like a funnel, first choosing the best tasks for generating candidate moves and then filtering, ranking and evaluating this set of promising moves to obtain an improved solution.

Candidate Task Identification

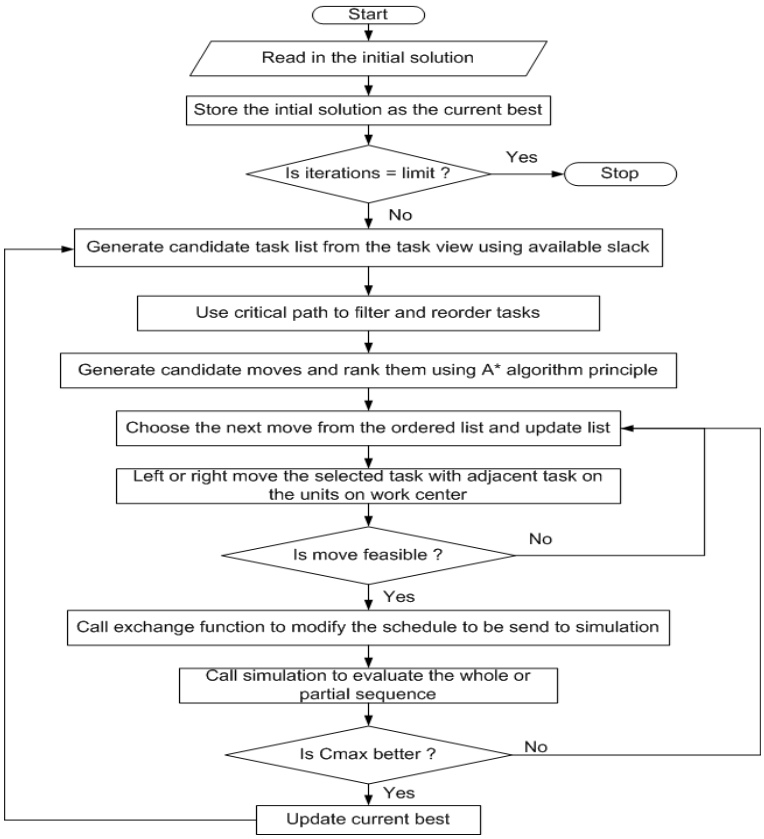

Figure 15: Flow of logic in SBLIMS algorithm

SBLIMS depends heavily on the candidate task list, which is the initial ordering of the tasks, used to generate candidate moves. The candidate task list consists of all the tasks that have slack before them in the current schedule's task view. In other words, the tasks have idle time following their immediate predecessors on the same job. It is worth noting that any task can have earliest start time greater than or equal to the latest finish time of its predecessor. This is evident from Theorem 4 in Chapter 2. The same cannot be said about resource constraints, which are bidirectional. An arbitrary amount of idle time can be inserted before or after any

task without violating the resource capacity constraint according to Theorems 2 and 3 in Chapter 2. This directional property is demonstrated with the help of an example. The same example is used to show how the candidate task list is generated during each iteration of SBLIMS algorithm.

The example has four jobs (1, 2, 3, 4), has four machines or work centers (A, B, C, D). Table 1 shows the route and processing times. For the time being assume that the setup times are included in the processing times.

Table 1: Example problem route

Job	Route Information	Processing Times (Mins)
1	A, B, C, D	5, 3, 7, 2
2	B, D, C, A	8, 6, 2, 5
3	B, D, C, A	3, 6, 4, 4
4	A, C, B, D	2, 2, 7, 3

Since we are using simulation to generate an initial schedule, we generate a random non-delay schedule as the initial solution. A random non-delay schedule can be defined as: *A schedule is said to be random non-delay schedule if the next task to be processed on any given work center is selected randomly from a list of given tasks waiting in the queue of the same resource under consideration as soon as a resource becomes available*Baker [26]. The advantage of the random non-delay schedule is that it guarantees that the initial solution is shifted left as far as possible. This implies that there is no place in the schedule where both the resource constraint as well as precedence constraint is simultaneously loose.

The random non-delay schedule generation was implemented as a dispatch rule in the simulation. When a resource becomes available, the next task to be processed on any given work center is selected randomly from the queue for that work center. Pseudo code is shown in Figure 16 for the random dispatch rule implementation.

1) Find the number of tasks (ntask) waiting in the resource queue;
2) Generate a random number between 1 and ntask;
3) Position at the beginning of the queue;
4) Traverse the queue until the required position is reached;
5) Retrieve the task from the queue;
6) Update the queue list;
7) Start processing the retrieved task;

Figure 16: Random dispatch rule logic

Any patterns created by the arrival times or sequence dependent setup times are broken using this random initial solution. Also, it is possible that a job that has an arrival time greater than another job at the same work center might be finished before the job that preceded it. This is analogous to a situation where a particular job has to be finished before another partially processed job. The only difference is that the random non-delay schedule has no control in the occurrence of such dynamics. Figure 17 illustrates the random non-delay schedule in both machine view as well as task view using Gantt bar charts.

There is an important difference between resource constraints and precedence constraints. Resource constraints are evident from the machine view of the schedule,

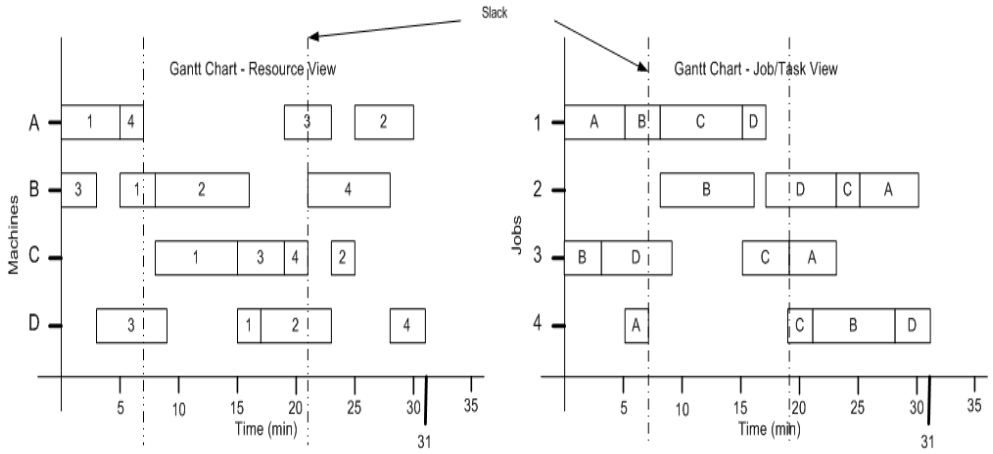

Figure 17: Non-delay schedule for the illustrative example

where idle times between tasks represent the "potential" for improvement. Machine constraints are bidirectional implying that a left or right shift of a task without violating capacity limit of machine is guaranteed to maintain the feasibility of the constraint. The precedence constraints on tasks are strictly directional. A shift can occur only in that direction where some slack exists for the task, either ahead (immediately after finish time) or before (immediately before start time). The successor or predecessor of the task constrains its degrees of freedom on shifts, which makes the precedence constraints more restrictive in nature.

Hereafter the job that has the task deciding the completion time will be named C_{max}-job. In this example it is job 4. Also job 4 has three of its tasks in the critical path: 4/C, 4/B, 4/D with the 4/D finish time equal to C_{max}. This task is known as C_{max}-task.

From Figure 17, in task view, 4C can be shifted left only until the start time of 4C is equal to the finish time of 4A; its predecessor. Meanwhile in the resource view, task 4 on machine C (equivalent of 4C in task view) can be shifted right or left as long as it avoids overlapping with other tasks, which still guarantees the resource constraint to be feasible. With this insight, the following statements can be considered as axioms about the candidate task list generation procedure.

Axiom 1 : *The schedule generated by simulation based on random dispatch rule on the machine queue generates a schedule with no free slack available to left shift and improve the objective function.*

Explanation : From the definition of a non-delay schedule, a machine is never idle when its queue is not empty. This implies that any task T_{ij} can be processed if:

- Task-i is present in the queue of Machine-j.
- A unit of Machine-j is free or capacity is available for processing.
- The selection rule identifies Task-i as the next task to process.

So any left shift of a task to reduce the idle time on a machine will result in right shifting of other tasks on the machine to maintain precedence feasibility. The example given in Figure 17 explains the phenomena.

Axiom 2 : *The C_{max}-job will have at least one of its tasks on the critical path of the schedule.*

Explanation : Makespan (C_{max}) of any feasible schedule is the longest path length (critical path) from start node to end node. Hence, to minimize makespan a feasible schedule needs to be found with minimum critical path length. A job by definition is the C_{max}-job if and only if the finish time of any of its tasks is equal to the maximum completion time, or the job contains the C_{max}-task. Since the C_{max}-task finish time determines the completion time, it is a node in the critical path.

Axiom 3 : *Only the tasks on the C_{max}-job with slack preceding it in the task view need to be considered for generating the moves in order to improve the makespan.*

Explanation : Consider a simple case of 4 machines (A, B, C, D) and jobs (P,Q). Figure 18 given below shows a feasible schedule with slack to be removed. It is obvious that the slack before job P on machine C can be used, whereas slack before job P on machine B and job P on machine D cannot be used because their precedence constraints are tight. Any shift will result in violation of the precedence constraints for these tasks of job P on the machines B and D.

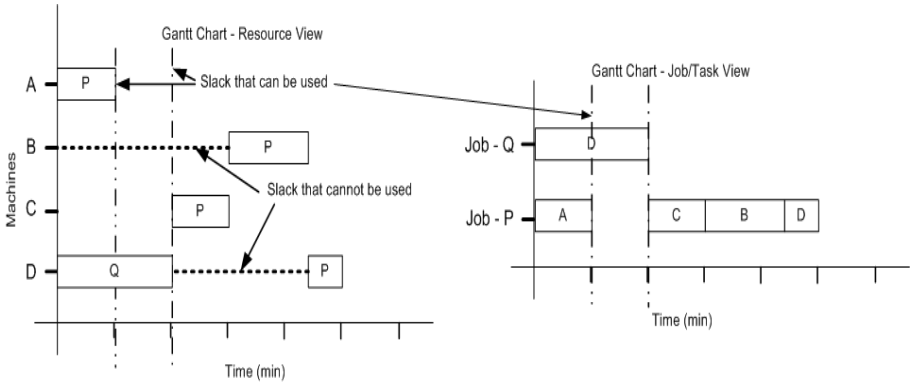

Figure 18: Usable and unusable slack in both task and resource views

Hence the existing slack is given by T_{QC}Start - T_{QA}Finish. This implies that the task that can reduce the makespan in this case is task-QC of job Q. This can be extended to conclude that useful slack can be more easily identified from the dual (task view) schedule than from the primal (resource view) schedule. So by alternating between two schedule views to identify tasks based on C_{max}-job using dual schedule and finding the moves based on the identified tasks using primal schedule can yield good results.

Now applying these principles to the example problem given in Figure 17, we conclude that the maximum completion time or makespan is 31 minutes. It is evident from the task view that job 4 has the maximum influence on the completion time, or the last task of job 4 is also the end node of the critical path. To show this, the network diagram of the job shop is shown in Figure 19 below. Also the critical path is marked on the diagram, to show the tasks on critical path, which are candidates for improvement. This critical path illustration is applicable to each reentrant loop in RFJSSDS problem. The critical path is evident in the task view itself too.

Axioms 1, 2 and 3 illustrate the principles upon which the candidate task list generation of SBLIMS algorithm is built. The next subsection describes about how the candidate task list is generated. At first we identify the task or tasks that have slack between the predecessor task from the task view. It is known from the shop scheduling literature that in order to minimize the maximum completion time, tasks on the critical path need to be rescheduled to other task positions, making use of

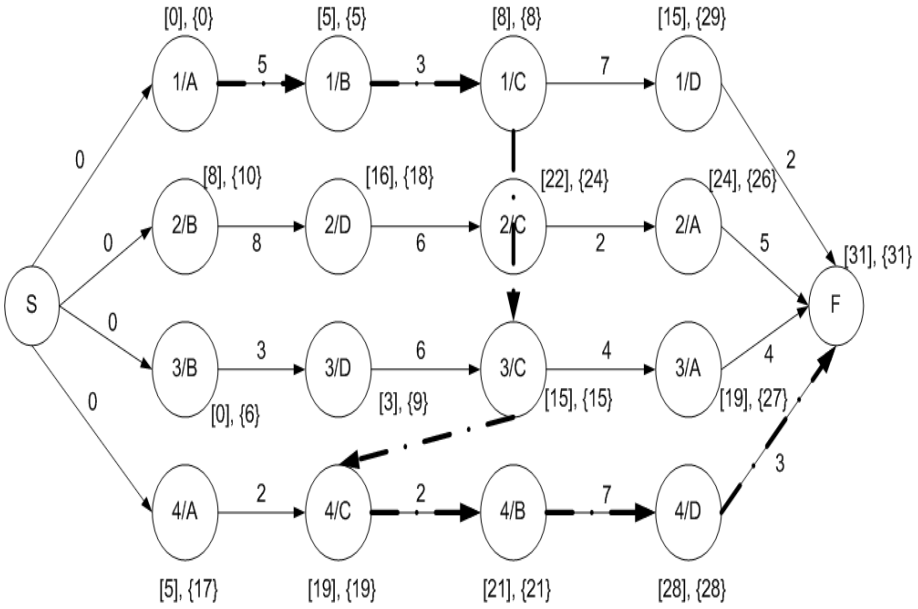

Figure 19: Critical path diagram for the example

any slack existing between these tasks and their predecessors. The candidate task list generation algorithm, which is a part of the search strategy, has a quick way to identify tasks on the critical path that belong to the C_{max}-job.

The proposed algorithm works with the *dual schedule* (task view) of RFJSSDS. The resource view of the schedule is referred to as the *primal schedule* in this search procedure. To illustrate the working, the example problem shown in Figure 16 is used. Pseudo code of the algorithm is given in Figure 20.

From the example given in Figure 16, the candidate list will contain the tasks given by {C4}; which in the primal view translates to job 4 on machine c. The tasks in the candidate task list are then used to generate the moves and rank them according to their estimated promise of improvement.

```
FOR all tasks in the Cmax-job, starting from Cmax-task DO
 IF (there is idle time immediately before current task) THEN
  IF (current task is not the first task of the job) THEN
   ADD task to the Candidate Task List;
  ELSE
   Obtain the next task and iterate;
END FOR loop;
RETURN Candidate Task List;
```

Figure 20: Candidate task list generation algorithm

Filter and Rank Moves

Having formalized the idea of candidate task list and its benefits, now we can look to the next step in the SBLIMS algorithm; the candidate move list. Once the candidate task list is created using available slack and critical path information, the tasks on the lists are used to create candidate move list. We know that SBLIMS algorithm perturbs the solution using the combination of left and right moves. This list contains moves that are promising as well as non-promising. By using the problem structure, we can reduce the number of possible moves to be evaluated, and even disregard quite a few without evaluation. Since only a small subset of moves can be evaluated due to time limitations, we filter and rank the candidate move list to obtain a better subset.

Move Filtering Criteria-1: Slack Window

Using the example given in Figure 17, we can identify a time window, that constrains the move options. Figure 21 defines that window for the example.

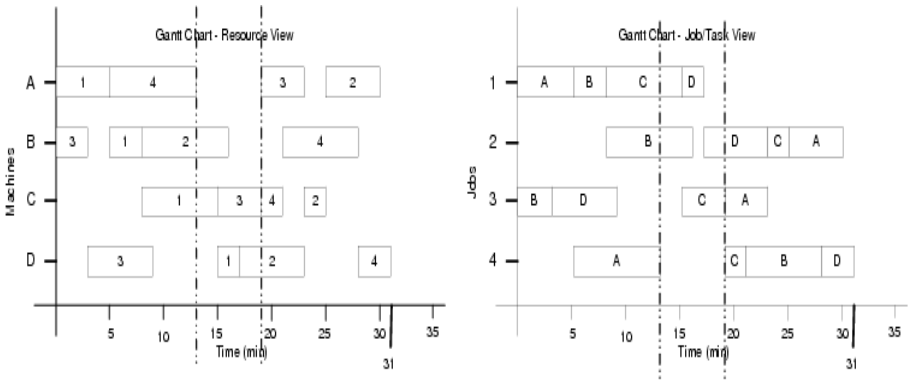

Figure 21: Time window for the given example

In this example, we can see that there are tasks on machine C that have start times before the time window. The nice thing about the time window is that we do not need to consider moves involving those tasks that have a start time before the predecessor of the task at some other machine, because that will eventually guarantee a worse completion time than the current one. This is an important axiom, which needs to considered first in ranking moves.

Axiom - 1 : *Only those moves involving a task from the candidate task list with other tasks on the same work center need to be considered, where the second task in consideration has a start time greater than or equal to the finish time of the predecessor of the task from the candidate task list on some other work center.*

Argument : Consider the case of job shop problem shown in Figure 21. Job 4 on machine C can be inserted before job 3 or job 1. It is also evident that job 1 has a start time less than job 4 finish time on machine A, which is the predecessor of job 4 on machine C. When an insert is performed before job 1, the precedence constraint

of job 4 gets violated. So to maintain feasibility we need to right shift the whole schedule from that point by a time value given by ((Start time of job 1 on machine C) - (Finish time of job 4 on machine A)). This implies that if the slack shown in the task view is lesser than the difference obtained above, chances are high that we might not be able to obtain any improvement and in the worst case we end up increasing the current maximum completion time by whatever the difference between slack and the time shift to maintain feasibility.

The example schedule takes the shape given in Figure 22 if the insert of job 4 on machine C is performed before job 1 on the same machine. In fact it is evident that the C_{max} increased by 6 minutes; which is greater than $(13 - 8 = 5)$ of the current feasibility shift to maintain the precedence feasibility.

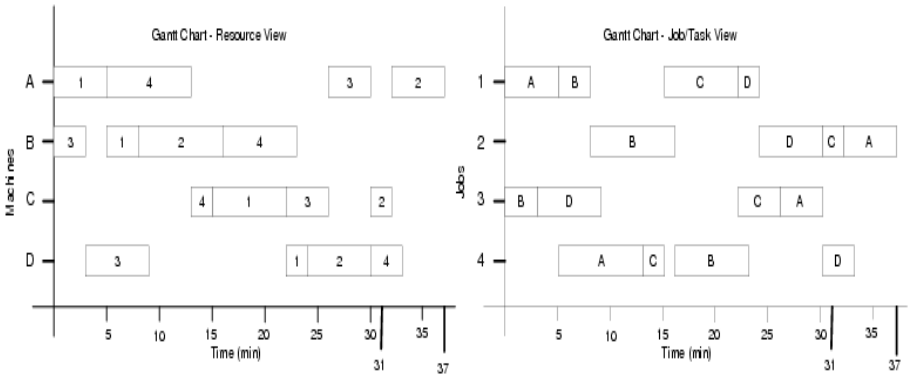

Figure 22: Insert of a candidate list task before another task on same machine

In a simple job shop problem we could find counter examples of this statement. But when the situation is to schedule a RFJSSDS with reentrancy loops, multiple machines, multiple products and sequence dependent setups, the chances of improve-

ment decrease rapidly because there are more jobs and hence more tasks. The result is more interactions and precedence constraints to be satisfied. Therefore it can be safely stated that the tasks that have a start time less than the predecessor of the candidate list task need not be considered for generating moves for the RFJSSDS problem.

Move Filtering Criteria-2: Reentrancy Limit

The other criteria that limits the choice of moves for the task from the candidate task list is the reentrancy limit. From Chapter 2, Theorem 1 states that an infeasible schedule will result if a reentrancy constraint is violated. Hence, in the case of the reentrant job shop, we need not look any further behind (to the left) the task list than the previous task of the same job on same machine in the previous loop. This is illustrated in Figure 23.

Move Ranking Criteria: A* Move Ordering

Once the lists of moves is obtained the moves are sorted according to their magnitude of promise towards improvement. A* is a form of best first search, which evaluates nodes by combining the cost to reach the node g(n), and the cost to get from the node to the goal h(n). The value associated by the node can be expressed as a function given by the $f(n) = g(n) + h(n)$ [27].

An important feature of the A* search is that it orders the nodes based on the value of f(n) associated with each node. Local search can be viewed as a tree search

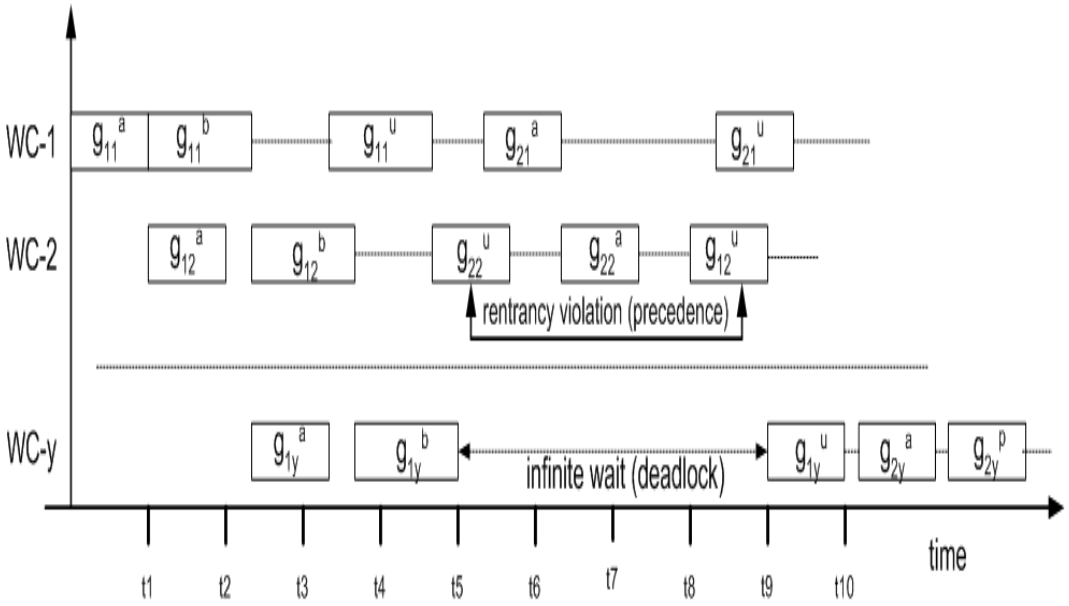

Figure 23: Violation of reentrancy constraint resulting infeasibility

where only a portion of the whole problem tree is evaluated. This property is adapted for the RFJSSDS problem. The estimation function value can be modified to be of the new form given in Equation 3.1. This function uses the property of the problem as well as the utility of the simulation to estimate the improvement.

$$\text{Improvement(move)} = \text{Current time slice (move)} - \text{Updated time slice (move)} \quad (3.1)$$

We argue that the improvement index is admissible because the maximum possible reduction in C_{max} without violating precedence constraints is given by the time slice in the given schedule. Figure 21 shows that the time slice is restricted by the precedence constraints. This is equivalent to the *Manhattan Distance* heuristic used in A* algorithm. The candidate move list is then sorted based on the improvement index and the move with the maximum value of improvement index is selected as

the move to perform and the schedule is updated. By the property of the A* search algorithm that if the heuristic estimate is admissible then the ranking is appropriate. An admissible heuristic estimate is always less than or equal to the true improvement.

Theorem 5 (Admissible Heuristic). *With an admissible heuristic for the move improvement index, the task ordering will represent the true ranking of moves.*

Proof :

Given that the heuristic is admissible, let us assume the following.

$$MS = \begin{cases} \text{a suboptimal move, with a completion time } C_{max}(s), \text{with} \\ \text{initial time slice value } ts_i(s) \text{ and final time slice value } ts_f(s). \end{cases}$$

$$M* = \begin{cases} \text{the optimal move, with the completion time } C_{max}*, \text{with} \\ \text{initial time slice value } ts_i* \text{ and final time slice value } ts_f*. \end{cases}$$

From Equation 3.1, we have:

$$Improvement(MS) = TSi(s) - TSf(s)$$

$$Improvement(M*) = TS_i * - TS_f*$$

By using the formula $f(n) = g(n) + h(n)$, the following can be concluded.

$$f(MS) = C_{max}(s) + Improvement(MS)$$

$$f(M*) = C_{max} * + Improvement(M*)$$

We know that $C_{max}* < C_{max}(s)$, because of the suboptimal goal assumption and since the heuristic never overestimates the improvement, we can say that $f(M*) < f(MS)$. So if all the moves are sorted based on the decreasing value of magnitude of improvement, then the list will represent the true order of improvement of the moves selected.

Once the moves are filtered and ranked, we evaluate them using time-slice simulation. The details of the time-slice evaluation are explained in the next section.

Simulation Model

This section describes a tool for simulating the generated problem instances of RFJSSDS. The instances of RFJSSDS are provided by the problem generator, which will discussed in the next chapter. Simulation is the most general approach to solve a complex system numerically.

The simulation tool was developed in such a way that any given instance of RFJSSDS can be simulated. The inherent generality of simulation was exploited to develop a unique and general approach in solving the RFJSSDS, which would then result in a general algorithm for solving such complex shops. The simulation tool was based on next event logic and was coded in C++. The simulation was developed in such a way that it will work as a simulator providing initial solution that gets used in a local search as well as a method to evaluate solutions during local search. The basic version of the simulation for reentrant flexible flow shop was obtained from Mooney and Kerbel [1]. The flow diagram of the simulator tool is given in Figure 24.

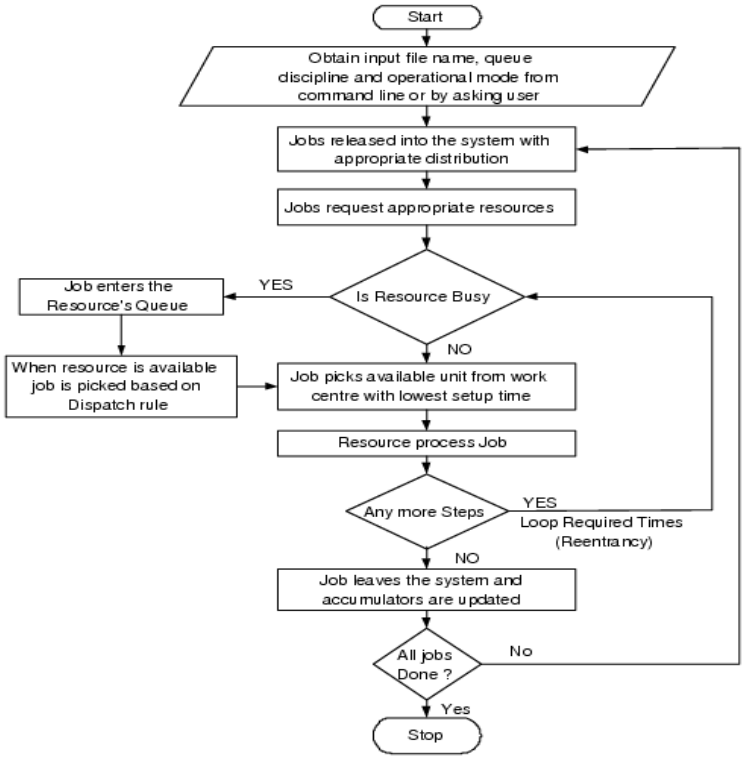

Figure 24: RFJSSDS problem simulator logic

The complete source code of the simulation and its related objects is available in Appendix B. The simulation uses these objects to represent the RFJSSDS model. The objects used in the simulation tool are:

- Job - stores the details of a particular job type in RFJSSDS.
- Task - each Job is a collection of tasks.
- Queue - the buffer space for each resource where tasks wait for getting processed.
- Resource - the work centers of RFJS where the operations are performed.

The initial version of the simulation was capable of simulating the system using five dispatch rules: Static Priority, First In First Out, Maximum Time in System, Critical Ratio and Least Slack. The simulation was first modified to accommodate the Random Rule to generate random initial solutions for this study.

The simulation creates arrivals based on the inter arrival times for each job. The complicated part of the RFJSSDS model is that all jobs are not available at the start of the simulation. The random arrival times restricts the options and choices for move selection during the local search, which will be discussed later. Job has routes that are specified by the input to the model. Based on the route associated with each job, the tasks are assigned to their respective work centers. So when a job arrives at a work center it can do one of the following two things:

- If the required unit of resource is free, the job can start getting processed.
- If the required resource is busy, the job has to wait in the resource queue.

Once the resource is free, it pulls the next task waiting in the queue based on the specified queuing discipline, provided there are tasks waiting. Otherwise the resource waits for the next task to arrive and the time is counted as idle time. Once a task has more than one unit of resource available at its disposal to begin the processing, the choice is made based on the unit of resource that will incur least setup time for the current task. The procedure continues until all the tasks are processed. The finish time of the last task to get processed is the maximum completion time; C_{max}.

CHAPTER 4

PERFORMANCE EVALUATION

This chapter documents SBLIMS performance on RFJSSDS and its restrictions. SBLIMS performance was evaluated using random instances of RFJSSDS as well as published problem instances for three important restrictions from the literature: classical job shop, reentrant flow shop, and flexible job shop. The random problem instances for RFJSSDS were obtained using a new problem generator. Results for RFJSSDS were compared with dispatch rules and GRASP [27] like multi-start alternative. Comparisons were made using instances specified from a preliminary study, identifying input input factors and the resulting experimental design. The SBLIMS algorithm was also tuned during this pilot phase of the study.

The organization of the rest of the chapter is as follows. We begin by describing the problem generator that was used to generate the RFJSSDS instances. Next we summarize RFJSSDS results, followed by the results of classical job shop, reentrant flow shop, and flexible job shop in that order. At the end of this chapter we conclude on the performance of SBLIMS on all the above class of shop scheduling problems. We begin by describing the problem generator

RFJSSDS Problem Generator

The RFJSSDS problem generator was used to obtain synthetic problem instances for the study. The generator was coded in C++. The problem parameters were

provided in a formatted data file to the generator. The parameters for the generator were passed via command line. All the data files are stored in a common directory and problem instances are generated in a single batch using bash scripts. The generated problem instances are stored in a separate directory. Each generated problem instance is named uniquely to identify problem characteristics from the name itself. These instances are then used by the simulation program to generate the initial random non-delay schedule. The local search described in Chapter 3 uses the initial schedule and generated problem instance together for improvement.

The data file was used to specify various aspects of the test problem instance like product route, sequence dependent setup times, processing times, reentrancy loops, etc. The flow of logic in the problem generator is depicted in Figure 25.

The generator allows specifying problems with a limited number of routes. For RFJSSDS each job can have a different route so the number of products can be the same as jobs. So the data file contains the information about the cumulative distribution of products. All jobs that need to be processed fall into one of the product categories. So the product mix array is generated using the cumulative distribution, which determines the category of product to which each job belongs to. A job structure is also created to store information like the number of operations, number of reentrancy loops and route information. The sequence for each product (job) to follow in the shop floor was determined using the generated route information. This models the real world implementations of RFJSSDS close enough, so that the

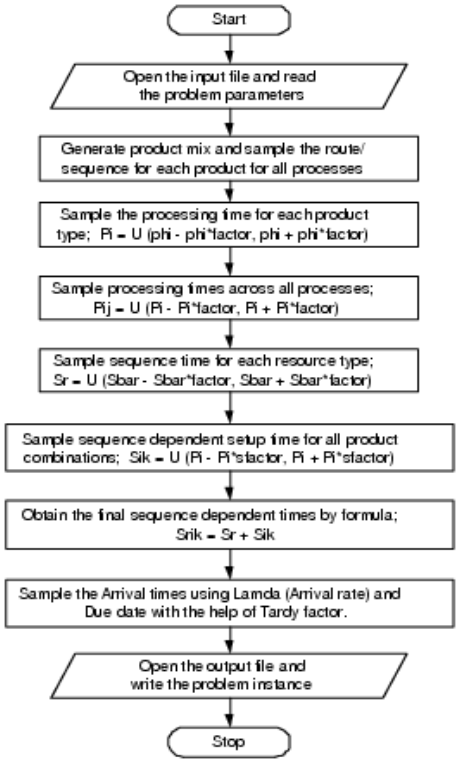

Figure 25: Problem Generator Logic

underlying structure in the shop floor is captured in the test problems.

The routes are obtained by the process of rejection sampling by generating a random number between the range of one and number of resources. A random number is generated and tested to be within the specified range. If such a number is found, then all previously assigned values in the route array are checked to verify whether the resource number is already present for the current pass. If so the value obtained is discarded, and a fresh number again sampled. This continued till a route for a pass is generated. Then the same route is copied for the rest of the reentrant loops, ensuring that the jobs follow the same route throughout. The pseudo code for the

route generation as explained above is given in Figure 26.

```
For all jobs for the given test problem do {
  For all operations for each product do {
    Generate a random number between 0 and number of resources;
    For all operations for the current product do {
      If resources visited before discard the sampled random number;
    }
     Decrement the operations index;
  }
}
```

Figure 26: Route generation procedure

Now we explain how process times were obtained. This is accomplished as a two-step sampling. The input file provides the long run average process time for all products in the shop. This value is in an uniform distribution whose upper and lower limit is constrained by user input process time deviation values. A random number is sampled, checked for being in the range of upper and lower limit of this distribution. This sampled time value is then used to generate the processing times for all individual processes that the product has to go through. The sampled time becomes the mean of the uniform distribution and the lower and upper deviations are obtained based on the process deviation factors available from input file. This uniform distribution obtains the individual processing times for all processes across a product type. The total time is calculated as a summation of the generated times. The total time is then stored in the product structure. This process is repeated across all product types. The Figure 27 summarizes the processing time sampling procedure.

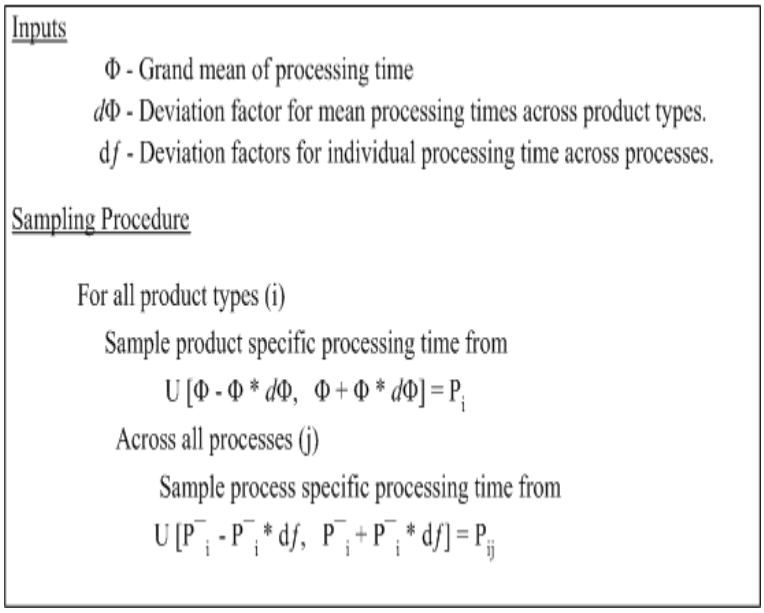

Figure 27: Processing time sampling procedure

The problem generator generates the sequence dependent setup time matrix for each resource. Sequence dependent setup times are quite common in the actual implementations of RFJSSDS and hence incorporated in this synthetic test problem generator. Sequence dependent setup times can be seen as a time value consisting of two parts. They are the resource specific part and the product specific part. The sequence dependent setup times are sampled using a two-step sampling procedure. The first sampling obtains the portion of the setup time associated with the resource. The time is sampled from a uniform distribution whose mean is obtained as the product of the grand mean of processing times with the setup severity factor. This setup severity factor has been defined in M. Pinedo [11]. The limits of the distribution are determined by the low and high deviation factors specified in the data file for each

resources. The first sampled time value provides the resource specific component of the sequence dependent setup times. The product specific component of the setup times get sampled next. The uniform distribution has the limits as the setup times for the individual products involved. This sampled values gets added with the resource specific time and the final total duration of sequence dependent setup times are obtained. The Figure 28 summarizes the sampling procedure for sequence dependent setup times.

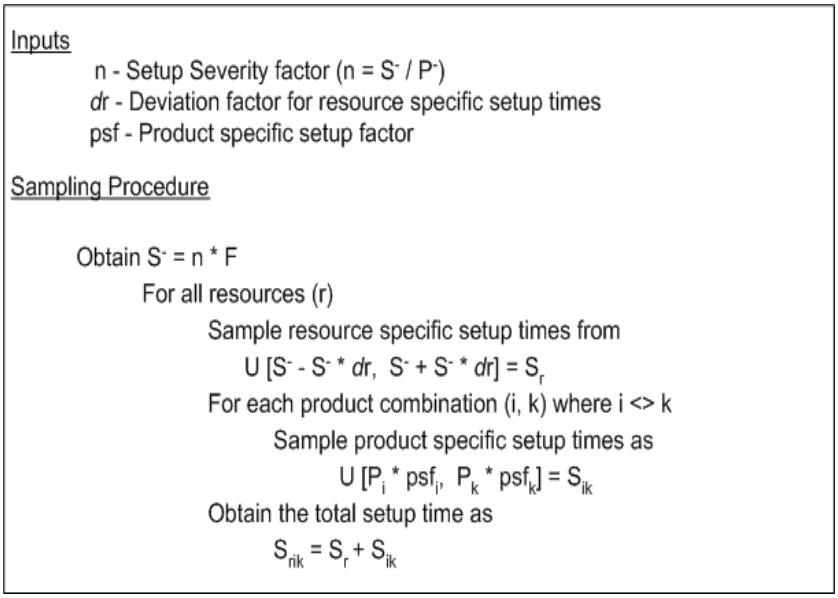

Figure 28: Sequence dependent setup time sampling procedure

Once the processing and setup times are allocated for each product, we obtain the Arrival times and Due dates. The arrival rate is calculated form the shop load factor and the service rate. The calculated arrival rate is then used as the mean of the exponential distribution to sample the arrival times. This approach gives a control over the arrival times as explained in Sule [28]. The pseudo code involved in

calculating the arrival rate is summarized in Figure 29. The equations states clearly that if the shop load factor is known, then the arrival rate can be calculated using the processing and setup times for the particular instance of the problem.

$\rho = \lambda / \mu$ where

ρ – shop load factor

λ – arrival rate

μ – service rate

μ consists of two components, the processing times and setup times. This implies $\mu = (\varphi + \sigma)$ where

φ – Sample mean of processing times

$$= \sum_{i}\sum_{j} P_{ij} \Big/ (\#products \times \#process)$$

σ – Sample mean of setup times

$$= \sum_{r}\sum_{i}\sum_{k} S_{rik} \Big/ (\#resources \times \#products)$$

So the final equation becomes;

$\rho = \lambda/(\varphi + \sigma)$ or in other words $\lambda = \rho/(\varphi + \sigma)$

Figure 29: Arrival times sampling procedure

The due dates are obtained as a function of total processing time inclusive of setup time, scaled with the tardy factor, which decides how much percentage of the jobs are late. The total processing time and setup time are known for a particular instance of the problem by summing all the individual samples of processing and setup times. Then the due date is obtained based on the pseudo code given in Figure 30. This approach of due date generation is used because by assigning random due dates, the chances of meeting the commitments are highly reduced.

All of the procedures mentioned above contribute to the successful generation of a test problem instance for RFJSSDS. The underlying mathematical principles used in

Due dates for any product (i)

$$D_i = (TP_i + TS_i) * (1 - \tau)$$

Where

D_i - calculated due date for product (i)
TP_i - Total Processing time for product (i)
TS_i - Total Setup time for product (i)
τ - Tardiness factor (or percentage of jobs tardy)

Figure 30: Due date sampling procedure

the problem generation attempt to capture the behavior of jobs in a typical RFJSSDS successfully. This makes sure that the generated test problems are not just random instances, but randomly generated instances capturing the underlying structure of the typical RFJSSDS. The source code and further details for the problem generator is given in Appendix A. The generated problem instances are used by the SBLIMS algorithm for the study.

RFJSSDS Results

The SBLIMS performance evaluation process consisted of the following steps:

- Tune the algorithm (by setting various parameters) using a small set of problems.
- Identify algorithms for comparison.
- Design the experiment and generate test instances.
- Finalize the factors from the screening experiment and run a an experiment

(using a 2^k factorial design) with a fresh set of test problem instances.

Each evaluation step is documented in the sections that follow. Results are summarized with tables. Details are given in Appendix C.

SBLIMS Algorithm Tuning

The tuning of the algorithm where various algorithm parameters are set is an important phase in any experiment. These parameters are important because they affect the performance of the algorithm. The SBLIMS algorithm parameters of interest are:

- Length of the candidate task list.
- Number of multi start points.
- Maximum run length for each starting point.
- Early termination criteria.
- Random number seed.
- Number of moves to be evaluated from the A* ordered list of moves.

An initial set of RFJSSDS problem instances were generated for tuning SBLIMS. These instances were generated using the problem generator. The tuning instances were not used in the experimentation phase of the study. The parameters for the RFJSSDS problem are summarized in Table 2.

Table 2: RFJSSDS problem parameter list

Parameter	Parameter Name	Description
1	jobs	Stores the number of jobs
2	products	Stores the number of products types (routes - 0 ⇒ unlimited)
3	setupsevfact	Value of setup time for resources
4	setupdeviation	Deviation from mean setup time - resource
5	mixarray	% of jobs that belong to each product
6	loadfactor	How heavily the shop is loaded (Rho)
7	phi	Grand mean of process times for the shop
8	prdfactordev	Deviation from phi for products
9	procfactordev	Deviation for processing time - processes
10	psfactor	Product setup time as % of its process time
11	ops	Number of operations
12	loops	No of loops (reentrant loop number)
13	capacity	Capacity at each work cell
14	tardyfactor	% of jobs that are tardy

The first algorithm parameter to tune was the number of multistart points SBLIMS Algorithm will generate during the search of one problem instance. This was set to 20 multistart points based on the study on the tuning instances. The reason was that the algorithm produced the best results for all tuning problem instances within fifteen random starting points. So a factor of safety of five more random starting points was added. Thus the final number of random non-delay starting points were limited to 20.

The second parameter to set was the length of the candidate task list. As mentioned during the discussion on the SBLIMS Algorithm, the candidate task list holds promising tasks on various work centers, based on the slack available before them, in the task view of the current feasible schedule. Longer the candidate task list,

more moves need to be examined. Since the tasks in the candidate task list are ordered based on the slack available immediately before them in the schedule, the most promising tasks will naturally appear at the top of the candidate task list. This was concluded from the explanations of Axiom 2 and 3 in Chapter 3. After running the SBLIMS Algorithm on the tuning instances the length of the candidate task list was finalized to be $\frac{N+M}{2}$.

Maximum run length was the third parameter addressed during the tuning phase. This is an important parameter for all liberated local searches where non-improving moves are permitted. Although SBLIMS is strictly an improvement algorithm where no non-improving moves are permitted, it was observed during the tuning phase that in RFJSSDS instances, there are lot of solutions that result in zero improvement that are permitted by the SBLIMS Algorithm. The algorithm often spends time traversing a plateau or ridge of the solution space. So an upper limit on the duration that could be spent on each starting point is a benefit. This parameter combined with the upper limit on zero improving moves (which is called as early termination criteria) limits the time for each multistart iteration. The maximum run length was set by the equation $0.1 * (\frac{N^2M+M^2N}{2})$.

The early termination criteria, the fourth parameter, discussed here determines performance of the algorithm in the solution time comparison. Though there is a maximum time limit for each starting point, the search can terminate in the following situations.

- When a selected move results in a C_{max} greater than the current best.
- If the last 50 consecutive moves produced solutions with zero improvement (or the C_{max} remained the same).

A random number seed, the fifth parameter, must also be specified for the SBLIMS Algorithm. Recall that the SBLIMS algorithm has a general purpose simulation embedded in it to perform functions like evaluation, initial solution generation, time slice simulation and updating the current solution. Like all stochastic simulation packages, sampling occurs here too and requires a stream of random numbers. Again the initial solution generated is a random non-delay schedule, where the task to be processed next get sampled from the respective queues of the work centers. So the random number seed should be controlled and kept constant through out the experimentation process.

The final parameter to be set is the number of moves to evaluate from the ranked move list after the A* ranking. All the moves that have an estimated improvement value greater than or equal to 50 percent of the best ranked move were selected for further evaluation. By evaluating a subset of moves, the algorithm is sped up. The risk is that we might lose an opportunity for an improvement, but the probability associated with such a loss is very small. The supporting evidence for this claim comes form Theorem 5 which states that the A* ordering heuristic is admissible.

Algorithms for comparison

The five algorithms used for comparison are summarized in Table 3.

Table 3: List of compared algorithms on RFJSSDS domain

Algorithm	Algorithm Name	Description
1	EDD	Earliest Due Date
2	SPT	Shortest Processing Time
3	LSLACK	Lowest Slack
4	Best Random Starting Point	Best from 100 Random Initial Solutions
5	SBLIMS Algorithm	Simulation based Local Improvement Multi Start Algorithm

The first three algorithms are dispatch rules. These rules are adapted from the single machine scheduling literature [29], [32], [33]. One of the most widely used one is the Earliest Due Date (EDD) rule. The developed schedule is based the due date of the jobs, with the job having the earliest due date scheduled first. This process is repeated until all jobs are scheduled. The Shortest Processing Time works the same way as the EDD rule using the smallest processing time as the job selection criteria. The final schedule has jobs is ascending order of their processing times.

The Least Slack Rule is another one pass heuristic. The next job to be scheduled is the job with least slack. The setup times and run times for all remaining operations are summed up first. This sum is then subtracted from the time remaining (now until the lot due date). This is called as the remainder slack and as mentioned before the job with the least slack get picked. This heuristic addresses the problem of work remaining.

The Best Random Starting Point (RSP) is a variation of the SBLIMS algorithm. This uses a next event simulation to evaluate the schedule but the jobs are picked at random from the queue of a work center. This continues until all jobs are finished and the schedule obtained is a random non-delay schedule. This process is repeated 100 times and the best random non-delay schedule generated is picked. The SBLIMS algorithm was explained in detail in the previous sections of this document and it is the final version of the algorithm.

Computational Experimental Design

After tuning the algorithm the next step was to conduct the experiment. To conduct the experiment, an experimental design is required. The experiment was conducted in two phases:

1. The algorithm ranking phase.

2. The significant problem parameter identification phase.

Since the same problems and results could be used for the two experimental phases, they were performed in conjunction. The section about problem generator mentions the various factors that are used in generating an instance of RFJSSDS. Table 2 summarizes all the factors that control the nature of any RFJSSDS problem instance created. Since not much research has been conducted on RFJSSDS, it is worth investigating the problem parameters influencing the solution quality. It is quite intuitive that an easier problem and a harder problem can be generated by providing

the problem parameters with different values. A perfect example of such a condition is having a problem where lot of good solutions are available, which will make the local search to reach a good solution faster. A reverse scenario would be to synthesize a problem with a lot of plateau like areas, where the local search spends more time traversing the neighborhood than taking improving moves.

From the fourteen factors listen in Table 2, a screening experiment was done to identify which factors were important to the study. An initial screening was performed based on the experience obtained while tuning the algorithm as well as running various restrictions of RFJSSDS problem. Of all the fourteen factors, some of them could be discarded right away, because the chances that these factors influence the algorithm performance is negligible.

The first one to be discarded is the setup deviation factor (parameter 4 in Table 2). This parameter is used to decide the upper and lower bounds of setup time sampling distribution. In the same way the product factor deviation and process factor deviation (parameters 8 and 9 in Table 2) were found insignificant. These factors decide the variability of processing time across products and process. In typical situations these deviations are at the most 10 percent of the mean processing time and do not vary that much. So for all practical purposes, we assume that these two stay constant.

A similar argument is extended to the factor named psfactor (parameters 10 in Table 2). The psfactor decides the sequence dependent setup time variability across the process for each product. In general we would expect the variation across products

to be minimal. Since the interest is to identify which factors contribute significantly to the character of the problem by changing its value, psfactor was dropped of because the variation in values for the parameter is quite small and negligible. So psfactor is kept constant in the experiment, so that by itself do not contribute to any variability.

The product mix array (parameter 5 in Table 2) is held constant too, because no products are given priority over any other (all the product types assume equal importance) and we do not limit the number of routes. So it can be viewed as another characteristic that can be studied separately in a situation where one product type has a precedence over other product type. This issue is outside the scope of this research and hence kept as a constant value throughout the experimental phase. The same can be said about the number of products. This factor decides how many product types the jobs belong to. Since there is no set priority among the products (routes) and no limit on the number of products, we can neglect this factor too.

This initial filtering leads us to a handful of factors for the study. The factors with their description and levels are summarized in Table 4.

RFJSSDS Results

The problem generator was used to generate three instances of each case in Table 4. The synthetic problems were grouped into three sets based on the number of jobs (N) and number of machines (M) pair. The three sets are summarized below:

1. Number of Jobs (N) = 5, Number of Operations (M) = 2.

Table 4: Fractional Factorial Experimental Design

Factor	Description	Factor	Values
N	Number of Jobs	-	5
		+	10
M	Number of Operations	-	2
		+	4
L	Shop Load Factor	-	0.3
		+	0.7
C	Number of Loops	-	2
		+	4
S	Capacity at work center	-	1
		+	3
P	Mean Processing Time	-	20
		+	40
T	Tardy Factor	-	0.3
		+	0.6

2. Number of Jobs (N) = 5, Number of Operations (M) = 4.

3. Number of Jobs (N) = 10, Number of Operations (M) = 4.

The $N = 10$ and $M = 2$ case was not included because they were just scaled problems of $N = 5$, $M = 2$ case. Table 5 through Table 8 summarize the results for the first set of synthetic RFJSSDS problems with their bounds gaps. As a reminder, the lower bound was calculated as the sum of all processing times and sequence dependent setup times. They are broken down into four smaller tables for easier reading based on problem difficulty. Table 5 shows the results for the easier problems in set 1. The shop load factor is 0.3, which implies that the shop is lightly loaded. The solution quality was measured as a percent deviation from the lower bound calculated.

Table 5: % Gap comparison for problem set 1: N=5, M=2, L=0.3

Features					Dispatch Rules			Best	SBLIMS
C	S	P	T	Rep	EDD	SPT	LSLACK	RSP	Algorithm
2	1	20	0.3	1	15.2	12.2	19.4	8.37	6.85
				2	16.9	12.8	20.3	8.43	6.67
				3	16.2	11.9	18.9	8.66	6.98
			0.6	1	32.5	28.5	34.2	12.78	7.13
				2	31.9	28.6	33.7	12.79	8.72
				3	32.7	29.4	34.5	12.12	9.13
		40	0.3	1	24.7	25.6	29.8	8.49	7.60
				2	25.6	25.9	29.7	8.66	8.22
				3	24.9	24.7	28.6	8.97	7.47
			0.6	1	24.1	19.2	27.8	17.23	16.26
				2	23.7	18.2	27.4	16.95	15.97
				3	23.9	18.8	26.1	17.02	16.44
	3	20	0.3	1	29.8	25.4	33.6	10.45	2.89
				2	29.4	25.7	33.9	10.97	3.67
				3	29.2	24.3	34.2	10.26	4.71
			0.6	1	19.2	17.7	24.7	10.61	9.52
				2	19.5	16.2	22.8	10.84	10.23
				3	18.9	16.6	23.1	10.03	9.26
		40	0.3	1	21.8	19.8	28.9	9.19	7.38
				2	20.9	19.1	28.1	9.47	7.21
				3	21.6	20.3	27.2	9.86	7.77
			0.6	1	24.4	20.7	29.5	10.61	9.52
				2	24.7	21.2	29.4	10.86	10.43
				3	23.6	21.4	29.9	10.24	10.21

Table 5 shows that the dispatch rules had almost consistent performance across the instances of the same problem. We can see that the Best RSP algorithm obtains a better solution in all the cases compared to dispatch rules. The SBLIMS algorithm developed to RFJSSDS domain outperforms all others in solution quality. The same problem parameters are kept for Table 6, which compares the solution quality in same problem category, but with four loops (C). So these problems are harder than the problems summarized in Table 5.

Table 6 also supports the claim that the SBLIMS algorithm performs better than dispatch rules and Best RSP algorithm. The harder problems of set 1 are given in Tables 7 and Table 8. These problems have the shop load factor of 0.7, making the

Table 6: % Gap comparison for problem set 1: N=5, M=2, L=0.3

Features					Dispatch Rules			Best	SBLIMS
C	S	P	T	Rep	EDD	SPT	LSLACK	RSP	Algorithm
4	1	20	0.3	1	22.1	20.1	25.7	12.45	10.41
				2	24.3	21.2	25.9	12.61	10.85
				3	23.9	22.4	26.1	11.97	9.71
			0.6	1	33.7	31.1	36.4	14.55	9.93
				2	30.4	31.6	36.3	15.67	7.65
				3	31.3	32.3	35.2	14.98	9.37
		40	0.3	1	14.3	16.4	20.3	10.44	7.63
				2	14.9	16.8	19.4	11.67	5.27
				3	13.6	14.3	18.5	11.83	5.84
			0.6	1	16.6	14.5	19.6	13.61	9.27
				2	17.8	15.9	20.4	12.22	9.91
				3	19.2	16.2	20.3	12.35	8.96
	3	20	0.3	1	34.9	32.9	37.1	16.91	10.78
				2	33.2	35.2	37.8	18.24	10.18
				3	33.9	34.7	33.4	17.66	10.62
			0.6	1	14.3	10.3	17.9	9.68	8.61
				2	14.8	12.2	19.4	8.74	8.22
				3	15.2	12.9	18.6	8.37	7.65
		40	0.3	1	18.4	18.1	21.1	12.33	8.21
				2	19.4	19.6	23.5	11.37	10.46
				3	17.9	19.9	22.7	11.89	7.14
			0.6	1	20.9	16.8	24.3	12.22	10.33
				2	20.6	18.1	25.7	13.46	10.21
				3	21.7	18.2	26.1	12.73	10.79

shop a heavily loaded. This could increase the interactions allowing less room for improvement.

Table 7 shows that the SBLIMS algorithm performing better than the dispatch rules and Best RSP algorithm. The difference between the bounds are less significant across algorithms because the shop is heavily loaded, and hence there is less room for improvement. Table 8 presents the summary of results for problem instances with same values of N, M, L, but with $C = 4$. Table 8 further demonstrates SBLIMS algorithm performs better compared to other algorithms.

Next we deal with the problem instances where $N = 5$ and $M = 4$. These instances are categorized as easy or hard based on the shop load factor, $L = 0.3$ or

Table 7: % Gap comparison for problem set 1: N=5, M=2, L=0.7

Features					Dispatch Rules			Best	SBLIMS
C	S	P	T	Rep	EDD	SPT	LSLACK	RSP	Algorithm
2	1	20	0.3	1	12.2	12.2	15.4	9.22	5.88
				2	11.1	12.8	16.3	9.57	6.24
				3	10.3	11.9	15.7	9.86	7.18
			0.6	1	14.2	13.6	15.2	10.34	8.64
				2	15.7	13.3	14.7	10.57	6.97
				3	15.4	13.1	15.5	10.85	7.28
		40	0.3	1	10.6	11.9	18.1	9.86	8.11
				2	11.3	10.4	17.7	9.54	8.06
				3	11.9	10.6	17.6	10.27	7.97
			0.6	1	14.7	10.9	14.8	9.65	6.26
				2	14.2	12.7	14.4	9.42	5.97
				3	13.8	13.2	15.2	10.13	6.44
	3	20	0.3	1	16.6	15.3	19.6	10.24	7.89
				2	16.8	16.4	19.9	10.63	6.67
				3	16.1	15.2	18.2	10.11	7.71
			0.6	1	12.1	13.1	18.7	10.68	8.52
				2	12.5	12.4	19.8	10.45	9.24
				3	12.8	11.6	17.1	10.03	8.27
		40	0.3	1	13.6	12.8	16.9	8.64	6.38
				2	13.7	11.7	15.4	9.48	6.12
				3	13.2	11.1	16.3	9.67	6.67
			0.6	1	11.8	12.4	16.4	10.66	9.02
				2	10.6	13.6	15.9	10.94	8.43
				3	11.3	11.4	15.7	11.08	8.21

$L = 0.7$. Table 9 consists of problem with parameters $N = 5, M = 4, L = 0.3$ and capacity $C = 2$.

For the same set of problem parameters, Table 10 summarizes the results with the number of reentrancy loops set to $C = 4$. More reentrant loops generate more tasks at each work center, thus making the problem size large.

From both Table 9 and Table 10 for problem set 2, we can see that SBLIMS algorithm performs better than dispatch rules and Best RSP. Best RSP comes close to SBLIMS in some instances in solution quality. We attribute this behavior due to the fact that the shop is not heavily loaded ($L = 0.3$), which results in shorter queues at the work centers. So the interactions are less and hence more flat areas in the

Table 8: % Gap comparison for problem set 1: N=5, M=2, L=0.7

Features					Dispatch Rules			Best	SBLIMS
C	S	P	T	Rep	EDD	SPT	LSLACK	RSP	Algorithm
4	1	20	0.3	1	18.6	20.1	22.4	14.47	11.48
				2	19.4	18.2	24.3	14.82	11.86
				3	19.7	19.5	23.9	13.73	11.71
			0.6	1	17.4	16.6	25.4	15.11	11.63
				2	20.3	19.3	22.6	14.63	10.64
				3	18.5	18.3	22.7	14.98	12.49
		40	0.3	1	15.4	16.3	21.1	12.15	10.44
				2	14.9	15.7	20.8	12.33	10.29
				3	15.8	14.9	21.2	12.65	11.31
			0.6	1	19.1	20.7	25.3	14.20	10.03
				2	17.6	20.4	22.1	15.07	11.92
				3	18.4	20.9	26.7	14.37	11.66
	3	20	0.3	1	21.3	22.6	27.5	16.09	12.78
				2	20.4	20.3	28.2	16.32	12.18
				3	20.9	19.8	26.8	15.88	13.62
			0.6	1	17.8	16.5	20.9	12.61	9.66
				2	16.7	19.1	22.6	12.34	9.07
				3	18.2	17.5	21.4	12.69	9.41
		40	0.3	1	18.4	18.1	27.8	11.44	8.83
				2	19.9	20.7	26.4	11.28	6.45
				3	17.1	16.3	26.3	11.89	8.97
			0.6	1	20.8	18.4	24.4	12.81	11.94
				2	22.6	20.3	24.9	11.04	9.07
				3	20.3	21.1	23.8	11.72	10.24

neighborhood, where the naive local search spends much more time traversing than improving the actual C_{max}.

In the following paragraphs, results for the harder problems with parameters $N = 5, M = 4$ are summarized. The shop load factor value of $L = 0.7$ makes the shop loaded heavily. Also the time at which the last job arrival to the system is added to the lower bound to make it more reasonable as explained before. Table 11 summarizes the problem instances with the number of reentrancy loops set to $C = 2$.

The same set of values for the problem parameters were used next, with the change in the number of reentrant loops, changed to $C = 4$. More reentrant loops means more tasks to process, which increases the problem size and difficulty. Table 12 summarizes

Table 9: % Gap comparison for problem set 2: N=5, M=4, L=0.3

Features					Dispatch Rules			Best	SBLIMS
C	S	P	T	Rep	EDD	SPT	LSLACK	RSP	Algorithm
2	1	20	0.3	1	11.6	11.8	16.5	9.37	8.83
				2	14.2	14.9	17.1	9.48	8.21
				3	12.9	13.3	17.3	10.27	8.64
			0.6	1	13.6	15.6	19.1	11.31	6.42
				2	15.7	14.7	19.9	11.09	9.21
				3	18.8	17.9	20.7	12.87	9.06
		40	0.3	1	14.6	15.4	20.3	11.33	8.44
				2	16.2	15.9	20.8	10.64	9.17
				3	15.4	15.6	20.6	10.71	7.32
			0.6	1	12.2	13.3	19.3	9.88	8.87
				2	13.7	13.2	18.2	9.94	8.46
				3	12.4	14.1	18.9	10.21	8.63
	3	20	0.3	1	19.2	20.7	27.1	13.37	6.78
				2	18.6	19.3	24.3	14.45	9.45
				3	18.1	19.9	25.7	14.93	8.97
			0.6	1	16.2	17.4	23.3	12.04	7.08
				2	16.6	16.6	25.1	11.31	9.34
				3	15.9	16.1	24.6	10.67	8.61
		40	0.3	1	19.6	20.3	26.6	12.43	10.37
				2	22.1	21.1	29.1	15.61	9.31
				3	21.7	21.4	27.9	14.88	10.08
			0.6	1	23.7	22.5	29.3	15.01	11.22
				2	24.8	22.6	28.9	15.66	10.35
				3	21.5	22.1	30.2	16.34	10.47

the performance of various algorithm on these RFJSSDS problem instances.

From Tables 11 and 12, we can see that again SBLIMS algorithm performs better on all test instances. It can also be noted that as the problem size increases, the SBLIMS algorithm shows a larger improvement against the dispatch rules and best RSP. We attribute this behavior to the fact that larger problem size in RFJSSDS provides more hills and valleys for the local search than flat plateaus.

The results for problem set 3 where the number of jobs is $N = 10$ are summarized next. The instances below have the problem parameter values $N = 10, M = 4, L = 0.7$. Table 13 shows results for the problems with the number of reentrant loops, $C = 2$.

Table 10: % Gap comparison for problem set 2: N=5, M=4, L=0.3

Features					Dispatch Rules			Best	SBLIMS
C	S	P	T	Rep	EDD	SPT	LSLACK	RSP	Algorithm
4	1	20	0.3	1	16.6	18.2	20.3	12.64	10.63
				2	17.8	16.5	20.1	11.03	10.21
				3	17.4	16.3	20.6	11.48	10.62
			0.6	1	15.5	14.1	19.7	12.21	10.66
				2	14.3	12.9	16.3	11.96	10.02
				3	14.8	13.1	18.4	11.64	10.31
		40	0.3	1	16.1	17.2	22.3	12.44	9.47
				2	16.6	15.4	21.3	11.93	9.25
				3	15.2	15.1	20.5	12.08	8.83
			0.6	1	19.1	19.5	27.4	13.27	9.61
				2	20.4	20.3	27.6	14.07	11.14
				3	21.6	20.8	25.5	15.31	10.05
	3	20	0.3	1	22.4	20.6	27.1	15.33	10.33
				2	19.6	20.7	26.7	16.91	10.95
				3	21.3	19.2	27.9	15.88	11.21
			0.6	1	16.3	16.5	20.9	13.61	8.64
				2	17.8	19.4	22.6	12.05	8.61
				3	18.1	17.5	20.4	11.35	9.13
		40	0.3	1	20.3	18.1	26.1	13.27	8.88
				2	20.7	20.7	26.3	13.28	9.63
				3	20.4	16.3	26.4	11.91	10.31
			0.6	1	18.6	17.6	25.4	14.25	11.22
				2	19.2	21.4	26.8	15.50	10.97
				3	18.5	20.7	24.1	15.03	11.04

The performance of various algorithms on the final category of test problems with number of reentrancy loops set to $C = 4$ is summarized in Table 14. The large number of jobs combined with the reentrant loops make the problem instances quite large.

From Tables 13 and 14, we can see that the SBLIMS algorithm outperforms the dispatch rules and Best RSP. The problem size is large and hence allows SBLIMS algorithm to have a larger candidate list. This increases, the number of choices, which translates to choosing better moves. So we can hypothesize that the neighborhood contains more good local optimums than the smaller problem.

Having seen how the SBLIMS algorithm provides better results in comparison with dispatch rules and Best Random Starting Points (RSP) across all problem instances,

Table 11: % Gap comparison for problem set 2: N=5, M=4, L=0.7

Features					Dispatch Rules			Best	SBLIMS
C	S	P	T	Rep	EDD	SPT	LSLACK	RSP	Algorithm
2	1	20	0.3	1	16.2	13.2	19.6	10.33	8.72
				2	15.9	14.1	21.3	10.93	8.67
				3	16.2	13.9	19.8	11.61	8.08
			0.6	1	22.5	20.5	24.2	12.78	8.13
				2	21.9	21.6	23.5	12.04	7.17
				3	22.7	20.8	24.9	11.62	8.65
		40	0.3	1	14.7	15.6	25.8	9.78	7.64
				2	15.6	15.9	26.1	10.21	8.20
				3	14.9	14.7	24.6	10.09	7.03
			0.6	1	24.8	21.8	26.8	14.32	10.12
				2	23.6	22.2	25.9	14.95	10.37
				3	23.1	21.6	26.1	15.02	9.86
	3	20	0.3	1	19.6	20.4	28.6	13.45	8.17
				2	19.4	20.7	26.4	12.72	8.58
				3	19.5	21.3	24.2	11.95	8.05
			0.6	1	15.2	16.7	19.8	10.61	7.52
				2	16.7	15.3	19.7	12.87	7.46
				3	17.9	15.6	19.1	11.03	8.12
		40	0.3	1	20.8	19.8	26.9	14.19	9.33
				2	21.9	19.3	25.7	12.47	10.91
				3	20.6	20.1	27.1	13.86	9.47
			0.6	1	25.4	24.7	29.5	15.61	10.61
				2	21.7	21.8	31.3	14.85	9.27
				3	20.6	20.1	30.2	15.64	10.68

we need to investigate the computational time requirements for these algorithms. We know that dispatch rules are one pass algorithms, which are constructive in nature and will generate a solution much faster, when compared to a local improvement algorithm. So to rank an algorithm, we need both the time efficiency as well as the solution quality.

The time taken for an algorithm from start to finish, including initialization, is considered as the running time of the algorithm. Here we have dispatch rules, that run under a fraction of a second. Also we have the local improvement algorithm which produces a better result in comparison with dispatch rules, but takes more time to complete the search. The computational study was conducted on a CPU with dual

Table 12: % Gap comparison for problem set 2: N=5, M=4, L=0.7

Features					Dispatch Rules			Best	SBLIMS
C	S	P	T	Rep	EDD	SPT	LSLACK	RSP	Algorithm
4	1	20	0.3	1	18.7	19.1	26.3	13.63	10.45
				2	20.4	20.6	25.1	12.21	9.85
				3	20.1	20.2	26.9	13.55	9.07
			0.6	1	23.5	21.9	29.4	17.64	11.93
				2	20.9	21.6	29.3	16.57	10.65
				3	21.4	22.4	29.5	16.93	12.33
		40	0.3	1	14.3	16.4	20.3	11.31	6.63
				2	15.9	13.8	21.7	11.04	7.25
				3	13.7	14.7	21.5	11.21	6.98
			0.6	1	16.2	15.5	19.6	12.14	9.42
				2	17.8	15.2	21.8	12.25	9.05
				3	19.6	17.1	21.3	12.65	8.64
	3	20	0.3	1	14.9	12.9	27.4	11.91	8.78
				2	13.2	15.5	27.8	11.24	8.68
				3	13.9	14.7	23.5	11.45	8.62
			0.6	1	24.2	25.3	27.9	18.64	12.61
				2	24.6	23.2	29.7	18.79	12.54
				3	25.2	24.9	28.6	17.37	12.65
		40	0.3	1	18.4	18.1	24.1	13.33	9.21
				2	19.8	20.6	24.5	13.07	9.65
				3	19.9	20.9	23.6	13.85	8.14
			0.6	1	24.9	26.8	31.3	16.82	12.53
				2	22.6	24.1	29.4	16.15	12.20
				3	26.7	24.2	31.8	15.05	12.19

3.06 GHz Pentium 4 processors, with Hyper Threading technology and 2 gigabytes of system memory (PC 166 RAM). All the reported times are CPU times. The time could be converted to CPU independent times using the coefficients specified in Dongerra [30].

For the problem category of N=5, M=2, L=0.3 and L=0.7 class problems the dispatch rules ran under fraction of a second. This is the easier class of RFJSSDS problems. The Table 15 below summarizes the time performance of various algorithms for RFJSSDS problem set 1, N=2, M=2, L=0.3 and L =0.7. For the exact CPU times, refer to Appendix C. For most instances the algorithm times were close to constant and hence could be represented as a $\bar{x} \pm s$.

Table 13: % Gap comparison for problem set 3: N=10, M=4, L=0.7

Features					Dispatch Rules			Best	SBLIMS
C	S	P	T	Rep	EDD	SPT	LSLACK	RSP	Algorithm
2	1	20	0.3	1	26.1	23.6	29.8	17.13	11.75
				2	25.9	24.1	31.6	16.47	10.43
				3	26.4	23.3	29.3	16.61	12.97
			0.6	1	22.5	20.5	34.2	15.78	9.43
				2	21.8	24.6	33.5	16.43	9.71
				3	22.3	24.7	34.9	17.82	8.05
		40	0.3	1	24.7	25.6	35.7	17.78	11.64
				2	25.6	25.8	36.1	15.25	10.25
				3	24.5	24.7	34.3	15.39	11.73
			0.6	1	24.8	26.9	36.8	14.32	10.19
				2	23.6	23.2	35.2	16.65	10.57
				3	24.1	26.6	36.1	16.72	10.86
	3	20	0.3	1	29.6	29.4	38.2	21.45	12.97
				2	29.2	29.5	36.4	21.74	12.68
				3	29.3	28.3	37.2	21.05	12.05
			0.6	1	25.4	26.7	29.2	18.68	13.52
				2	26.7	25.3	29.7	18.32	9.86
				3	27.9	25.6	29.6	17.64	10.92
		40	0.3	1	20.8	22.7	26.9	14.19	10.57
				2	21.1	21.3	25.7	13.55	10.89
				3	20.6	22.3	27.8	13.91	9.67
			0.6	1	25.4	24.6	31.5	17.31	11.34
				2	24.3	26.9	32.8	16.47	9.86
				3	23.5	25.4	31.4	17.84	11.24

For problem set 2 where N=5, M=4, L=0.3 and L=0.7, similar behavior was noted. The time performance for those instances is summarized in Table 16. These are CPU times from the test machine, not CPU independent times.

From Table 15 and 16, we can see that the SBLIMS algorithm takes more time to obtain a solution. However the time invested in choosing the best move pays off as better solution quality.

The problem instances given by parameters N=10, M=4, L=0.3 and L=0.7 behaved differently compared to the previous sets. Since these instances are larger, all algorithms took more time to obtain a solution. Table 17 summarizes the time performance of the competing algorithms. For the exact values of the CPU time (in

Table 14: % Gap comparison for problem set 3: N=10, M=4, L=0.7

Features					Dispatch Rules			Best	SBLIMS
C	S	P	T	Rep	EDD	SPT	LSLACK	RSP	Algorithm
4	1	20	0.3	1	28.7	29.1	36.8	17.47	11.51
				2	30.4	30.6	35.1	17.24	10.75
				3	30.3	30.4	36.9	16.05	11.15
			0.6	1	33.5	31.9	39.4	21.83	12.83
				2	30.9	31.6	39.3	20.55	10.79
				3	31.4	32.2	39.9	18.12	13.54
		40	0.3	1	34.1	36.4	40.3	21.69	10.63
				2	35.8	33.8	41.7	21.07	12.41
				3	33.7	34.3	41.5	22.27	10.98
			0.6	1	26.2	25.5	39.1	16.14	10.42
				2	27.9	25.2	31.8	17.42	11.65
				3	29.6	27.5	31.3	19.56	12.12
	3	20	0.3	1	34.4	32.9	37.4	21.19	11.78
				2	33.1	35.5	37.8	21.42	11.43
				3	33.9	34.7	33.5	21.89	10.62
			0.6	1	34.2	35.7	37.9	23.55	12.17
				2	34.6	33.2	39.4	21.63	13.54
				3	35.2	34.9	38.6	21.37	11.65
		40	0.3	1	28.9	38.1	40.1	19.43	10.26
				2	29.8	30.7	42.8	18.03	11.32
				3	29.9	30.9	41.6	19.85	12.18
			0.6	1	34.4	36.8	41.1	21.18	12.61
				2	32.6	34.1	39.2	23.95	10.93
				3	33.7	34.4	41.6	21.43	10.44

seconds) for any instance of the problem, refer Appendix C.

From Table 17, SBLIMS algorithm takes more time than all other competing algorithms in generating a solution. Also the time increases proportionally more than the other algorithms. The extra time invested on computation translates to a better solution quality in all the cases, as shown in Tables 5 through 14. So dispatch

Table 15: Solution time summary for set 1: N=5, M=2, L=0.3/0.7 problems

Algorithm	Algorithm	CPU Times(sec)	
	Name	Average	Std. Dev.
1	EDD	0.031	0.004
2	SPT	0.033	0.002
3	LSLACK	0.049	0.009
4	Best RSP	0.145	0.035
5	SBLIMS	0.465	0.061

Table 16: Solution time summary for set 2: N=5, M=4, L=0.3/0.7 problems

Algorithm	Algorithm Name	CPU Times(sec) Average	Std. Dev.
1	EDD	0.037	0.002
2	SPT	0.041	0.003
3	LSLACK	0.043	0.003
4	Best RSP	0.191	0.021
5	SBLIMS	0.512	0.043

rules provide us with a quick solution, but of the least quality. If we can invest little bit more time to generate some random solutions and pick the best out of it (Best RSP), we get better solution quality. If sufficient time can be invested to perform the local search, we obtain better solutions.

Having seen the solution quality and the time taken for reaching the solution, we used a program for ranking the algorithm. This ranking program is named as ANLYZ, and was developed by Mooney [31]. The program was modified for the RFJSSDS problems and the algorithms that needs to be compared. Each of the competing algorithms got a Z_{rank} for solution quality and T_{rank} for time to reach the best solution. The details of the modified source code and outputs are available in the Appendix C. Figure 31 summarizes the solution quality and time rank in a $0-1$

Table 17: Solution time summary for set 3: N=10, M=4, L=0.3/0.7 problems

Algorithm	Algorithm Name	CPU Times(sec) Average	Std. Dev.
1	EDD	0.587	0.031
2	SPT	0.603	0.038
3	LSLACK	0.561	0.051
4	Best RSP	1.237	0.183
5	SBLIMS	12.512	2.733

scale, as reported by the ANLYZ program.

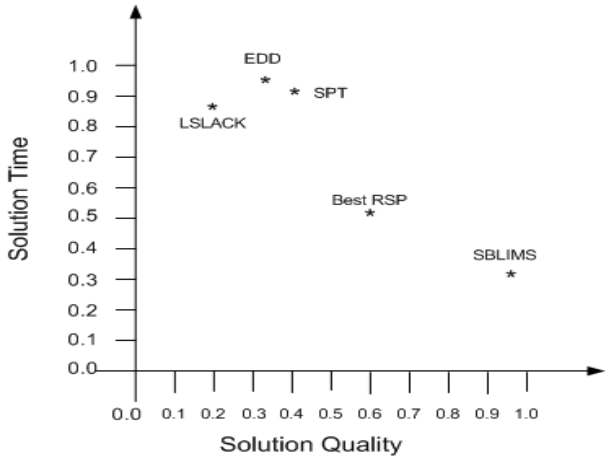

Figure 31: Algorithm rankings for RFJSSDS problems

Figure 31 shows that the SBLIMS algorithm ranks better in solution quality but worst with respect to computation time among most instances. The ideal algorithm would be the one that dominated all others in both dimensions on all instances and would be plotted on the upper right corner. The best RSP algorithm gives the best tradeoff in both time and quality dimensions. The dispatch rules form a cluster by themselves as expected, with smallest computation times and worst makespan for most instances.

We have seen that the SBLIMS algorithm performed better on the RFJSSDS problems. An important motivation for the development of SBLIMS algorithm was the generality behind the approach. SBLIMS algorithm can be applied to various restrictions of RFJSSDS problems, namely classical job shop problems popularly known as $J//C_{max}$ problems, Reentrant flow shop (RFS) and Flexible job shop (FJS). Re-

sults and comparisons of the SBLIMS algorithm performance for these problems are presented in the following sections.

Classical Job Shop Results

The test problems used in this study are obtained from Applegate [32]. The classical Jobshop problems are usually denoted by $J//C_{max}$ notation. As discussed in previous sections these problems are NP-Hard [11]. The problems discussed in the Applegate paper are obtained from various sources:

- All problem files with names 'abz' are by Adams et.al [33] via Applegate.
- All problem files with names 'car' are by J. Carlier via Applegate.
- All problem files with names 'la' are by S. Lawrence [34], via Applegate.
- All problem files with names 'mt' are by Muth and Thompson, via Applegate.
- All problem files with names 'orb' are by Bonn in 1986, via Applegate.

As mentioned above each of these categories were solved with the SBLIMS algorithm developed for RFJSSDS problems. The SBLIMS algorithm used 20 random starting points which are non-delay schedules and improved on that. The results shown here is the best obtained over all the 20 starting points. The percentage improvement over each problem category will be summarized after the discussion on solution quality and solution time. All solution times are on a dual Intel pentium processor at 3.06 gigahertz clock speed, with 2 megabytes of system RAM. We sum-

marize the CPU times for the Best RSP and SBLIMS algorithm for all class classical job shop problems. In addition we compare SBLIMS and Best RSP times with the ten machine problems belonging to the 'la' problem class.

The algorithms are compared to the best known and optimum values of these test problems. The GRASP-like Best RSP algorithm generates random non-delay schedules and pick the best schedule from them. The number of random non-delay schedules generated was limited to 100. The algorithms were applied to the problems provided by Applegate and the results are summarized in Tables 18 through 24.

The first column in each table gives the problem name, the second column shows the number of jobs (N) and number of machines (M). The best known column gives either the optimal solution or best known lower and upper bounds.

The Best RSP columns in Tables 18 through 24 shows the results for the random dispatch rule. The Best RSP algorithm constructs 100 random non-delay schedules and the best one out of the 100 is picked. The Gap subcolumn in the Best RSP column denotes the percent gap with the best known solution for the given problem, with CPU times shown in the adjacent subcolumn.

The SBLIMS algorithm column has the results summarized in three subcolumns. The Improve subcolumn shows the improvement from the starting point to the best C_{max} found by the local search. The second subcolumn shows the solution gap obtained by SBLIMS with best known C_{max} or bound for the problem. The last subcolumn shows the CPU time taken by SBLIMS to solve a problem instance. Table 18

summarizes the results for Adams et.al. [33] class of job shop problems.

Table 18: Solution Quality of SBLIMS algorithm on 'abz' problems

Prob.	Specs.		Best Known	Best RSP		SBLIMS Algorithm		
Name	N	M		Gap	Time(sec)	Improve	Gap	Time(sec)
abz5	10	10	1234 (Opt)	3.81	0.04	5.23	0.567	0.52
abz6	10	10	943 (Opt)	6.57	0.06	10.33	0.636	0.64
abz7	20	15	(654, 668)	(20.95, 18.41)	0.09	15.14	(5.05, 2.84)	1.39
abz8	20	15	(635, 687)	(30.08, 20.23)	0.07	18.34	(9.92, 1.60)	1.71
abz9	20	15	(656, 707)	(31.25, 21.78)	0.09	20.03	(9.60, 1.69)	1.83

We can conclude from Table 18 that SBLIMS algorithm have gaps less than 1.83 percent, which is close to optimum. Also it takes around ten times more time to obtain that solution when compared with Best RSP. Next we summarize the results for Carlier problems in Table 19.

Table 19: Solution Quality of SBLIMS algorithm on 'car' problems

Prob.	Specs.		Best Known	Best RSP		SBLIMS Algorithm		
Name	N	M		Gap	Time(sec)	Improve	Gap	Time(sec)
car1	11	5	7038 (Opt)	6.19	0.09	6.06	0.128	0.91
car2	13	4	7166 (Opt)	16.61	0.12	16.31	0.251	1.27
car5	10	6	7702 (Opt)	4.96	0.06	4.69	0.856	1.84
car7	7	7	6558 (Opt)	16.89	0.02	16.72	0.152	0.61

Similar conclusions can be reached from Table 19, where SBLIMS algorithm have gaps less than 1.84 percent. But Best RSP algorithm ran faster, and the gap values were less than 16.89 percent.

The SBLIMS algorithm results for problems from Lawrence et.al. are summarized in Tables 20 through 22. The problem set is divided into three subsets for 5, 10 and 15 machines. The 5 machine problems are on the easier side, whereas the others are harder. The makespan results for 5 machine subset are summarized in Table 20.

As in the cases above a set of random solutions were generated and compared with SBLIMS algorithm.

Table 20: Solution Quality of SBLIMS algorithm on 'la' problems (5 m/c)

Prob.	Specs.		Best Known	Best RSP		SBLIMS Algorithm		
Name	N	M		Gap	Time(sec)	Improve	Gap	Time(sec)
la01	10	5	666 (Opt)	0.002	0.21	2.24	0.405	2.41
la05	10	5	593 (Opt)	0.000	0.28	6.24	0.000	2.73
la06	15	5	926 (Opt)	0.000	0.43	0.0	0.000	3.68
la07	15	5	890 (Opt)	9.10	0.39	9.17	0.449	3.12
la08	15	5	863 (Opt)	9.50	0.46	9.50	0.000	3.47
la09	15	5	951 (Opt)	0.000	0.41	0.0	0.000	4.19
la10	15	5	958 (Opt)	0.000	0.48	0.31	0.000	4.47
la11	20	5	1222 (Opt)	0.081	0.61	2.37	0.000	6.31
la12	20	5	1039 (Opt)	4.62	0.72	6.35	0.000	5.92
la13	20	5	1150 (Opt)	1.91	0.68	2.87	0.087	5.84
la14	20	5	1292 (Opt)	0.000	0.63	0.0	0.000	6.43
la15	20	5	1207 (Opt)	11.76	0.74	11.76	0.000	6.22

From Table 20 we can see that SBLIMS algorithm found the optimum in nine out of the twelve instances. The time taken by SBLIMS algorithm was around ten percent more than the time taken by Best RSP algorithm. It is worth noting that the Best RSP also found optimum solutions in five out of the twelve instances.

The next subgroup of Lawrence problems had 10 machines with varying job numbers. This category of the problems is considered hard in most job shop related literature like Vaessens, Aarts and Lenstra [26], and Applegate and Cook [31]. So these problems were studied separately as a measure for the SBLIMS algorithm. The results are summarized in the Table 21.

From Table 21 we conclude that the percent gap for SBLIMS algorithm was less than 5.47. Also the SBLIMS algorithm took more time to obtain the solutions in comparison to Best RSP. The Best RSP algorithm had percent gap values less than

Table 21: Solution Quality of SBLIMS algorithm on 'la' problems (10 m/c)

Prob.	Specs.		Best Known	Best RSP		SBLIMS Algorithm		
Name	N	M		Gap	Time(sec)	Improve	Gap	Time(sec)
la16	10	10	945 (Opt)	11.64	0.47	10.45	2.33	3.45
la17	10	10	784 (Opt)	12.37	0.36	12.79	1.66	2.97
la18	10	10	848 (Opt)	13.68	0.41	14.47	1.06	2.81
la19	10	10	842 (Opt)	7.96	0.49	5.57	2.26	3.04
la20	10	10	902 (Opt)	24.28	0.53	23.02	3.55	2.95
la21	15	10	(1040, 1053)	(25.19, 23.65)	0.84	21.23	(5.19, 3.89)	3.67
la22	15	10	927 (Opt)	32.25	0.91	30.40	1.83	3.53
la23	15	10	1032 (Opt)	11.63	0.79	10.45	2.03	3.94
la24	15	10	935 (Opt)	19.79	0.96	17.77	1.71	3.77
la25	15	10	977 (Opt)	14.12	0.83	14.19	0.921	3.65
la26	20	10	1218 (Opt)	18.23	1.13	17.17	0.903	4.39
la27	20	10	(1235, 1269)	(26.72, 23.32)	1.22	18.38	(7.04, 4.18)	5.34
la28	20	10	1216 (Opt)	23.27	1.09	19.82	2.88	4.37
la29	20	10	(1120, 1195)	(20.71, 13.14)	1.04	11.19	(9.29, 2.43)	4.65
la30	20	10	1355 (Opt)	12.69	1.17	11.25	1.70	4.45
la31	30	10	1784 (Opt)	13.51	1.93	10.41	2.80	4.25
la32	30	10	1850 (Opt)	10.05	1.87	7.19	2.97	4.63
la33	30	10	1719 (Opt)	10.41	1.66	4.69	5.47	4.89
la34	30	10	1721 (Opt)	12.20	1.72	7.16	4.71	4.72
la35	30	10	1888 (Opt)	13.98	1.85	10.15	3.87	4.79

32.25. So SBLIMS algorithm obtained a better quality solution using the extra time it spend on doing the local search.

The final subgroup of Lawrence problems had 15 machines. These problems are considered the hardest. The SBLIMS algorithm and Best RSP results are summarized in Table 22.

Table 22: Solution Quality of SBLIMS algorithm on 'la' problems (15 m/c)

Prob.	Specs.		Best Known	Best RSP		SBLIMS Algorithm		
Name	N	M		Gap	Time(sec)	Improve	Gap	Time(sec)
la36	15	15	1268 (Opt)	20.34	1.48	17.96	3.63	5.94
la37	15	15	1397 (Opt)	17.54	1.32	14.64	3.65	5.63
la38	15	15	(1184, 1217)	(22.97, 19.64)	1.35	12.07	(10.55,7.56)	6.24
la39	15	15	1233 (Opt)	17.03	1.51	12.12	4.38	5.72
la40	15	15	1222 (Opt)	20.05	1.46	13.63	5.65	5.51

Similar conclusions can be made from Table 22, just like the 10 machine subset of Lawrence problems. Results for the Muth and Thompson problems are summarized

in Table 23. These job shop problems are generally considered to be moderately difficult.

Table 23: Solution Quality of SBLIMS algorithm on 'mt' problems

Prob.	Specs.		Best Known	Best RSP		SBLIMS Algorithm		
Name	N	M		Gap	Time(sec)	Improve	Gap	Time(sec)
mt06	6	6	55 (Opt)	7.27	0.03	0.0	0.000	0.46
mt10	10	10	930 (Opt)	20.43	0.24	18.27	1.83	0.83
mt20	20	5	1165 (Opt)	22.83	0.31	23.38	0.601	0.55

Final testing of the algorithm for job shop problems was done on Bonn problem set. The results of the testing is summarized in the Table 24.

Table 24: Solution Quality of SBLIMS algorithm on 'orb' problems

Prob.	Specs.		Best Known	Best RSP		SBLIMS Algorithm		
Name	N	M		Gap	Time(sec)	Improve	Gap	Time(sec)
orb1	11	5	1059 (Opt)	16.90	0.11	15.42	2.27	0.51
orb2	13	4	888 (Opt)	14.53	0.17	13.27	2.70	0.59
orb3	10	6	1005 (Opt)	26.07	0.13	24.71	3.88	0.43
orb4	11	5	1005 (Opt)	16.12	0.21	11.53	4.38	0.67
orb5	13	4	887 (Opt)	16.57	0.24	16.32	2.93	0.52
orb6	10	6	1010 (Opt)	15.45	0.18	11.90	3.17	0.48
orb7	11	5	397 (Opt)	12.85	0.25	10.44	3.78	0.42
orb8	13	4	889 (Opt)	22.16	0.16	16.89	4.49	0.47
orb9	10	6	934 (Opt)	11.24	0.12	9.05	2.57	0.39
orb10	10	6	944 (Opt)	21.08	0.09	18.94	1.8	0.65

From the five different classes of classical jobshop problem we can see that SBLIMS algorithm produced excellent results. While the best known solutions were found rarely, gaps were typically less than 5.85 percent with a few seconds of computing time. There are algorithms like shifting bottleneck [32], which run very fast, but the solution quality is inferior compared to the SBLIMS algorithm.

Additionally comparisons were made with three other contesting algorithms. The main contestants here are the shifting bottleneck algorithm, the tabu search and

random non-delay solution (Best RSP) algorithm. As expected the shifting bottleneck algorithm proposed by Applegate and Cook [32] was the fastest. Good solutions were also found in most cases. The Tabu Search algorithm by Dell'Amico and Trubian [35] gave best results reported for these problems. The tabu search is famous for its bidirectional search and uses the union of two neighborhoods [36].

The computer independent run time values were obtained from a study conducted by Vaessens, Aarts and Lenstra [36]. They used the eleven classical job shop problem instances shown in Table 25. All the instances are from Lawrence [34] and are generally considered among the most difficult of this size. Table 25 summarizes the findings. All the times given are CPU independent times computed using time factors from Dongerra [30].

Table 25: Comparison of algorithms on 'la' problems (10 m/c)

Prob.	Specs.		TS	SB		Best RSP		SBLIMS		
Name	N	M	Times (sec)	%Gap	Time (sec)	%Gap	Times (sec)	%Gap	Times (sec)	% over TS
la19	10	10	519	8.5	65	7.96	31	2.26	597	15.03
la21	15	10	994	11.6	46	23.65	29	3.89	1011	1.71
la24	15	10	909	9.3	63	19.79	32	1.71	1023	12.54
la25	15	10	958	6.7	48	14.12	30	0.921	912	-4.80
la27	20	10	1271	5.8	92	23.32	44	4.18	1114	-12.32
la29	20	10	1407	9.9	91	13.14	66	2.43	1302	-7.46
la36	15	15	1192	6.3	153	20.34	89	3.63	1008	-15.44
la37	15	15	1211	8.9	56	17.54	42	3.65	1106	-8.67
la38	15	15	1283	5.7	96	19.64	93	7.56	1039	-19.02
la39	15	15	1189	6.6	134	17.03	101	4.38	1042	-12.36
la40	15	15	1183	9.4	24	20.05	42	5.65	1002	-15.30

From Table 25, we can see that the shifting bottleneck algorithm runs much faster. The Tabu search provides the best solution quality with all the optimal or best known solutions. So the SBLIMS algorithm was compared against the machine independent

times of Tabu search, rather than SB algorithm as shown in the last column. We can see that the SBLIMS algorithm reports better times in most of the test problems. Also the Best RSP algorithm runs in comparable times to SB algorithm, but has inferior solution quality.

Reentrant Flow Shop Results

The Reentrant Flow Shop (RFS) is a restriction of RFJS problem. In RFS all jobs available at the start of the scheduling period. The shop has its resources, organized as a flow shop, where all jobs follow the same route. Jobs reenter the shop again for another set of processing after completing a pass through the facility. The job can skip some machines it visited in the previous loop. The processing times, setup times and due dates for each job is provided.

These problems have been studied by various authors. The pioneering study in RFS domain was done by Graves, Meal, Stefek and Zeghmi [4]. They were the first to explain the importance of the reentrancy in this problem. More research has been done in this field by Demirkol and Uzsoy [14], Yang, Kreipl and Pinedo [2], and Uzsoy, Lee and Martin-Vega [9].

In this section of the thesis, we extend the SBLIMS algorithm to handle the maximum lateness objective, which is denoted by L_{max}. The test problems were obtained from Ovacik and Uzsoy [15]. These problems were synthesized RFS problems and there are salient features associated with them. The processing times and setup times of the jobs follow a Uniform distribution between 1 and 200 [15]. Due dates also

follow a uniform distribution and the interval of the distribution is determined by the expected system workload and parameters R and τ, where τ denotes percentage of jobs that are tardy [14]. There are five machines with 10 or 20 jobs to be processed [15]. Table 26 summarizes the parameters for RFS problems obtained from [15].

Table 26: Parameter Summary of RFS synthetic problems

Parameter	Parameter Name	Values			
1	Due Date Range	0.5	1.5	2.5	
2	Percent Tardy Jobs	0.3	0.6		
3	Number of Machines	5			
4	Number of Jobs	10	20		
5	Number of Loops	1	2	4	6
6	Operations per Loop	U(1,3)	U(3,5)		

The following section summarizes the performance of SBLIMS Algorithm on selected problems. The SBLIMS algorithm was modified to handle a different objective function, maximum lateness L_{max}. The other major change was in selection criteria of the task list. The task selection was done for tasks with slack preceding them. But the ranking was based on which task can influence the L_{max}. So the tasks that will improve the maximum lateness or minimize the maximum lateness was ranked higher. With these modifications the algorithm was able to run on a RFS problems with a different objective function. Table 27 summarizes the results for *re305* class RFS problems. As an additional comment, we would like to state that this approach of simulation based local search is general enough to accommodate multiple objectives and problem characteristics.

Table 27: Solution Quality of SBLIMS algorithm on 're305' problems

Prob.	Specs.		Best Known	Best RSP		SBLIMS Algorithm		
Name	N	M	L_{max}	L_{max}	Gap	L_{max}	Improve	Gap
1	10	5	741	797	7.56	753	7.84	1.62
2	10	5	771	849	10.17	789	15.34	2.33
3	10	5	552	688	24.64	558	22.04	1.09
4	10	5	449	610	35.86	446	15.92	-0.007
5	10	5	547	602	10.05	578	5.36	5.67
6	10	5	703	812	15.51	719	18.92	2.28
7	10	5	457	538	17.72	451	27.94	-1.31
8	10	5	314	499	58.92	324	56.48	3.18
9	10	5	653	697	6.74	669	4.93	2.45

Flexible Job Shop Results

In classical job shop problem there are n jobs to be processed on m machines. By definition these machines are unrelated. Each job has its own route, depending on the operations that need to be done. Also all jobs and machines are available at time zero. There is no preemption, and the objective is to minimize C_{max}.

The flexible job shop (FJS) is an extension of the classical job shop, where there exists at least one instance of the machine type necessary to perform the operations [7]. There major contributors in the area is Chambers and Barnes [7], where they developed an adaptive tabu search method to solve the FJS problems. Since the FJS has more than one instance of a machine type, the problem becomes that of assigning each operation to the appropriate machine and sequencing the operations on each machine [7]. The test problems were generated based on the classical job shop problem instances. The chosen instances were MT10, LA24, LA40, with eight replications of each instances. Table 28 summarizes the results for MT10 class of FJS problems.

Table 28: Solution Quality of SBLIMS algorithm on 'MT10-FJS' problems

Prob.	Specs.		Best Known	Best RSP		SBLIMS Algorithm		
Name	N	M	C_{max}	C_{max}	Gap	C_{max}	Improve	Gap
p1	10	10	929	1002	7.86	931	9.67	0.24
p1,p1	10	10	929	1060	14.10	930	4.43	0.13
p1,p1,p1	10	10	936	1104	17.95	945	11.74	0.97
p1,p2	10	10	913	1044	14.35	922	16.61	0.98
p1,p2,p3	10	10	849	997	17.43	891	3.74	4.94
$c1^c$	10	10	927	1086	17.15	933	5.55	0.65
$c1^c$	10	10	928	1047	12.82	930	6.67	0.22
c1,c2	10	10	919	1063	15.67	941	8.97	3.26
c1,$c2^d$	10	10	914	1091	19.37	922	5.21	0.88

Similarly the results for LA24 FJS problem is summarized in Table 29. The instances were generated on the same logic. The first three instances denotes the two and three time replication of the machine with maximum number of critical jobs. From the fourth instance onwards the authors replicated different machines on the critical path.

Table 29: Solution Quality of SBLIMS algorithm on 'LA24-FJS' problems

Prob.	Specs.		Best Known	Best RSP		SBLIMS Algorithm		
Name	N	M	C_{max}	C_{max}	Gap	C_{max}	Improve	Gap
p1	10	15	937	1034	10.35	944	3.21	0.75
p1,p1	10	15	930	1071	15.16	939	6.84	0.97
p1,p1,p1	10	15	925	1063	14.92	942	5.87	1.84
p1,p2	10	15	924	993	7.47	937	11.31	1.41
p1,p2,p3	10	15	914	1131	23.74	931	7.49	1.86
c1	10	15	924	1004	8.66	951	9.67	2.92
c1,c2	10	15	926	1061	14.58	947	8.45	2.27
c1,$c2^c$	10	15	909	1044	14.85	929	6.67	2.21

Table 30 summarizes the results for LA40 FJS problems. The instances were generated based on the same logic as that of the LA24 and MT10 set of FJS problems.

From Tables 28 through 30, we can see that SBLIMS algorithm performs better in solution quality. The percent gaps are typically less than 4.94 percent.

Table 30: Solution Quality of SBLIMS algorithm on 'LA40-FJS' problems

Prob.	Specs.		Best Known	Best RSP		SBLIMS Algorithm		
Name	N	M	C_{max}	C_{max}	Gap	C_{max}	Improve	Gap
p1	15	15	1218	1264	3.8	1220	4.7	0.16
p1,p1	15	15	1204	1293	7.4	1210	11.2	0.49
p1,p1,p1	15	15	1213	1347	11.1	1221	14.7	0.65
p1,p2	15	15	1148	1217	6.0	1161	5.1	1.13
p1,p2,p3	15	15	1127	1196	6.2	1149	6.4	1.95
c1	15	15	1185	1263	6.6	1192	7.3	0.59
c1,c2	15	15	1136	1245	9.6	1139	11.9	0.26
c1,$c2^c$	15	15	1137	1205	5.9	1151	3.4	1.23

Summarizing this chapter, the SBLIMS algorithm performed well on RFJSSDS instances as well as the restrictions of RFJSSDS like classical job shop, reentrant flow shop and flexible job shop problems.

CHAPTER 5

CONCLUSIONS

This study presents a new simulation based local search approach for solving shop scheduling problems. Results for classical problems from the literature demonstrate the effectiveness and quality of the approach. Application to a new, very general class of problems, Reentrant Flexible Job Shop with Sequence Dependent Setup times (RFJSSDS) is provided as well.

The reentrant flexible job shop with sequence dependent setup times (RFJSSDS) problem is a generalization of the classical job shop, reentrant flow shop and flexible job shop problems. In RFJSSDS each work center can have more than one unit of resource and jobs can cycle through, or reenter, the shop multiple times until the required number of loops through the shop is completed. RFJSSDS is more general and difficult due to multiple products (routes), sequence dependent setup times at the work centers and reentrancy of the jobs.

The solution methodology developed in this study combines simulation with an innovative local search in an algorithm. The simulation is a very general object oriented next event shop model. The local search algorithm modifies an initial feasible solution provided by the simulation module to generate promising neighbor solutions. A generated solution is considered to be better if there is a reduction in the total completion time, or makespan. A unique candidate filtering strategy is used to select and rank moves. This strategy uses both task and resource views of a schedule to

select moves. Also incorporated into the algorithm is an approach motivated by A* search to rank moves. Multiple random starting points are generated in multistart fashion as part of the solution process. This Simulation Based Local Improvement with Multi Start (SBLIMS) algorithm exploits new results presented as theorems that provide insight to the RFJSSDS problem.

The algorithm was evaluated using published test problems for several well known shop scheduling problems as well as RFJSSDS. The RFJSSDS restrictions studied were classical job shop, reentrant flow shop and flexible job shop problems. The SBLIMS algorithm in all the cases provided excellent results compared with those provided in the literature.

A set of synthetic problems was also generated to study the RFJSSDS since there were no RFJSSDS problem sets available. The SBLIMS algorithm was compared against various dispatch rules in the RFJSSDS domain and its performance was found to be better most of the time and much better on relatively larger problems.

REFERENCES CITED

[1] E. Mooney, R. Kerbel, "Multiproduct Flowshops with Reentrant Jobs and Sequence dependent Setups", *Unpublished working paper*, Mechanical and Industrial Engineering Department, Montana State University, Bozeman, 2003.

[2] Y. Yang, S. Kreipl and M. Pinedo, "Heuristics for minimizing total weighted tardiness in flexible flow shops", *Journal of Scheduling*, 3:pp 89-108, 2000.

[3] E. Nowicki and C. Smutnicki, "The flow shop with parallel machines: A tabu search approach", *European Journal of Operational Research*, 106:pp 226-253, 1998.

[4] S. C. Graves, H. C. Meal, D. Stefek and A. H. Zeghmi, "Scheduling of Reentrant Flow Shops", *Journal of Operations Management*, Vol.4, No.4, pp 197-207, August 1983.

[5] S. Verma and M. Dessouky, "Multistage hybrid flowshop scheduling with identical jobs and uniform parallel machines", *Journal of Scheduling*, 2:pp 135-150, 1999.

[6] J. B. Chambers and J. W. Barnes, "Reactive search for flexible job shop scheduling", *Graduate program in Operations Research and Industrial Engineering*, The University of Texas at Austin, Technical Report Series, ORP98-04, 1998.

[7] J. B. Chambers and J. W. Barnes, "Tabu search for the flexible routing job shop problem", *Graduate program in Operations Research and Industrial Engineering*, The University of Texas at Austin, Technical Report Series, ORP96-10, 1996.

[8] M. Mastrolilli and L. M. Gambardella, "Effective neighborhood functions for the flexible job shop problem", *Journal of Scheduling*, 3:pp 3-20, 2000.

[9] R. Uzsoy, C.Y. Lee and L. A. Martin-Vega, "A review of production planning and scheduling models in the semiconductor industry part II: shop floor control," *IIE Transactions*, vol.26, No. 5, pp 44-55, Sept. 1994.

[10] E. Mooney, "Private Communication", *Mechanical and Industrial Engineering Department*, Montana State University Bozeman, August 2004.

[11] M. Pinedo, "Scheduling Theory, Algorithms and Systems", *Englewood Cliffs* New Jersey, Prentice Hall, 2002.

[12] S. J. Russell and P. Norvig, "Artificial Intelligence A Modern Approach", *Pearson Education Inc.*, Prentice Hall, 2003.

[13] B. Jeong and Y. Lee, "Performance analysis of lot release control and dispatching rules in semiconductor wafer fabrication", *Journal of Manufacturing Systems*, 17:pp 41-52, 1997.

[14] E. Demirkol and R. Uzsoy, "Decomposition methods for reentrant flow shops with sequence-dependent setup times", *Journal of Scheduling*, 3:pp 155 -177, 2000.

[15] I. M. Ovacik and R. Uzsoy, "Exploiting shop floor status information to schedule complex job shops", *Journal of Manufacturing Systems*, 13:pp 73-84, 1994.

[16] C. Y. Liu and S. C. Chang, "Scheduling flexible flow shops with sequence dependent setup effects", *IEEE Transactions on Robotics and Automation*, vol.16, No. 4, pp 408-419, August 2000.

[17] F. Y. Ding and D. Kittichartphayak, "Heuristics for scheduling flexible flow lines", *Computers in Industrial Engineering*, vol. 26, No.1, pp 27-34, 1994.

[18] V. J. Leon and B. Ramamoorthy, "An adaptive problem space based search method for flexible flow line scheduling", *IIE Transactions*, 29:pp 115-125, 1997.

[19] E. G. Negenman, "Local search algorithms for the multiprocessor flow shop scheduling problem", *European Journal of Operational Research*, 128:pp 147-158, 2001.

[20] S. A Brah and L. L. Loo, "Heuristics for scheduling in a flow shop with multiple processors", *European Journal of Operational Research*, 113:pp 113-122, 1999.

[21] R. L. Daniels, S. Y. Hua and S. Webster, "Heuristics for parallel-machine flexible resource scheduling problems with unspecified job assignment", *Computers and Operations Research*, 26:pp 143-155, 1999.

[22] C. S. Sung, Y. H. Kim and S. H. Yoon, "A problem reduction and decomposition approach for scheduling for a flowshop of batch processing machines," *European Journal of Operational Research*, 121: 179-192, 2000.

[23] S. J. Mason, and K. Oey, "Scheduling complex job shops using disjunctive graph: a cycle elimination procedure", *International Journal of Production Research*, Vol.41, No.5, pp 981-994, 2003.

[24] S. J. Mason, S. Jin, and C. M. Wessels, "Rescheduling strategies for minimizing total weighted tardiness in complex job shops", *International Journal of Production Research*, Vol.42, No.3, pp 613-628, 2004.

[25] P. Toth and D. Vigo, "The granular tabu search and its application to the vehicle routing problem", *Informs Journal on Computing*, Vol.15, No.4, pp 333-346, Fall 2003.

[26] K. R. Baker, "Introduction to Sequencing and Scheduling", *John Wiley Sons*, New York, 1974.

[27] S. Rojanasoonthan and J. Bard, "A GRASP for parallel machine scheduling with time windows", *Informs Journal on Computing*, Vol.17, No.1, pp 32-51, Winter 2005.

[28] D. R. Sule, "Industrial Scheduling", *PWS Publishing Company*, Boston, 1997.

[29] I. Ahmed and W.W Fisher, "Due Date assignment, Job Order Release and Sequencing", *Decision Sciences*, 23, pp 633-644, 1992.

[30] J. J Dongerra, "CPU independent time coefficients", *ACM Transactions on parallel computing*, Vol.3, pp 56-136, 2004.

[31] E. L. Mooney and R. L. Rardin, "Tabu search for a class of scheduling problems", *Annals of Operations Research*, Vol.41, pp 253-278, 1993.

[32] D. Applegate and W. Cook, "A computational study of the Jobshop scheduling problem", *ORSA Journal of Computing*, Vol.3, pp 149-156, 1991.

[33] J. Adams, E. Balas and D. Zawack, "The Shifting Bottleneck procedure for Job Shop Scheduling", *Management Science*, Vol.34, pp 391-401, 1998.

[34] S. Lawrence, "Resource Constrained Project Scheduling: an Experimental Investigation of Heuristic Scheduling Techniques (Supplement)", *Graduate School of Industrial Administration*, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1984.

[35] M. Dell'Amico and M. Trubian, "Applying tabu search for job shop scheduling problem", *Annals of Operations Research*, 41, pp 231-252, 1993.

[36] R. J. M Vaessens, E. H. L Aarts and J. K Lenstra, "Jobshop scheduling by local search", *Informs Journal on Computing*, Vol.8, No.3, pp 302-317, summer 1996.

[37] I. M. Ovacik and R. Uzsoy, "Rolling horizon algorithm for a single machine dynamic scheduling problem with sequence dependent setup times", *International Journal of Production Research*, Vol.32, No.6, pp 1243-1263, 1994.

[38] K. Horiguchi, N. Raghavan, R. Uzsoy and S. Venkateswaran, "Finite capacity production planning algorithms for semiconductor wafer fabrication facility", *International Journal of Production Research*, Vol.39, No.5, pp 825-842, 2001.

[39] H. Fisher and G. L Thompson, "Probabilistic learning combinations of local job-shop scheduling rules", in J.F Muth and G.L Thompson (eds.), *Industrial Scheduling*, Prentice Hall, Englewood Cliffs, New Jersey, pp 225- 51, 1963.

[40] C. Rajendran and H. Ziegler, "A performance analysis of dispatching rules and a heuristic in static flowshop with missing operations of jobs", *European Journal of Operational Research*, 131:pp 622-634, 2001.

[41] P. Schillings, "Private Communication", *Mechanical and Industrial Engineering Department*, Montana State University, Bozeman, August 2004.

[42] C. Rajendran and O. Holthaus, "A comparative study of dispatching rules in dynamic flow shops and job shops", *European Journal of Operational Research*, 116:pp 156-170, 1999.

APPENDICES

APPENDIX A - PROBLEM GENERATOR

This section describes the inner details and coding approach to the problem generator. The generator obtained the necessary input values for various parameters from a data file. These .dat files contain the input parameters that defines an instance of the RFJSSDS problem. The input to the generator followed the order given in Table 31.

Table 31: List of input parameters for the problem generator

Sl. No	Field Name	Data Type	Description
1	jobs	Integer	Stores the number of jobs
2	products	Integer	Stores the number of product types
3	setupsevfact	Float	Value of setup time for resources
4	setupdeviation	Float	Deviation from mean setup time - resource
5	mixarray	Float	% of jobs that belong to each product
6	loadfactor	Float	How heavily the shop is loaded (Rho)
7	phi	Float	Grand mean of process times for the shop
8	prdfactordev	Float	Deviation from phi for products
9	procfactordev	Float	Deviation for processing time - processes
10	psfactor	Float	Product setup time as % of its process time
11	ops	Float	Number of operations
12	loops	Float	No of loops (reentrant loop number)
13	capacity	Integer	Capacity at each work cell
14	tardyfactor	Float	% of jobs that are tardy

All the .dat files that store the information about various problem scenarios are stored in the directory named as "/problem". The generated instances of the problem are stored in the directory "/instances", where the generated instances had an extension of .prb. The initial part of the generator sets the path and obtains the name of the input file from the user. This name is then used to open the required problem scenario from the directory "/problem". After the .dat file is opened the values in the file is read in as in the order given in Table 31.

These values are then used to generate other necessary parameters. Before using these values all of them are printed out to make sure that the values are read in correctly. Once the integrity and continuity of the .dat files are established this section of the code was made as a debug utility for the problem generator for further modifications.

After reading the values, the first step is to generate the product mix array. This is accomplished by using the percentage of jobs belonging to each product information from the .dat file and cumulated to get the discrete distribution ranging from 0 to 1. The number of operations and loops are then stored in their respective variables in the product structure. A check is made to see whether the number of operations exceeded the number of resources. If such a case existed the number of resources are made equal to the number of operations, so that no shortage is experienced.

Values for maximum number of loops and maximum number of operations are also found out using a similar approach. A resource array is created next. The number of elements of this array is equal to the number of resources and each array element is populated with the capacity value, i.e. the number of machines available at each work cell. The product structure itself has a sequence array built into it that carries the sequence or route the product should follow.

The sequence dependent setup matrix was created as a part of a resource structure. An instance of this structure will be available for each resource and stores the sequence dependent setup matrix and other sampled values. By this way the efficient use of the limited static memory could be done.

Upon completing this, the number of tasks for simulation was calculated. This is obtained as the sum total of the product of operations and loops of each product type the job belongs to. After this the necessary problem parameters are stored in .prb file as output, which is to be used by the simulator to obtain the initial solution. The complete source code for the generator is given below. The generator was written in C++ and compiled using GNU C++ in a Linux machine.

```
/*----------------------------------- Generator.cpp --------------------------------------*/
// The program generates various problem instances of flexible flow shop with reentrancy
// and sequence dependent setup time effects
// Code written by : Deepu Philip
// Purpose of code : Master's Thesis
// Guided by       : Dr. Ed Mooney
//
// Initial code written in C++ on : 04/14/2004
// Variable declarations accomplished, input output files setup
//
// Modified on : 04/15/2004
// Dynamic memory allocation with new operator in C++
// Array of structures replaces the old multi dimensional arrays
// Sampling routines for processing and setup times
// Input file update, calculation of lamda, improve sampling procedure

#include <stdio.h>
#include <iostream.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>

#include "util.h"
#include "rutil.c"
#include "rutil.h"
#include "distributions.cpp"

#define SIZE 200
#define DETAIL FALSE

struct Productstr   // structure for product array
```

```
{
    float psfactor;         // product factor for setup times
    int opsno;           // Number of Operations
    int loopsno;           // Number of Loops
    int seqarray[SIZE+1];  // stores the machine sequence for a product - route
    int indpt;         // stores individual processing time sampled
    int totptime;      // stores total processing time
    int totstime;      // stores total setup times involving the product
    int ptimei[SIZE+1];    // stores the processing time across each processes
};

struct Setupstr     // structure for setup time matrix
{
    int indsetup;                   // stores the sampled setup time for resource
    int sdsetuptime[SIZE+1][SIZE+1];  // stores the setup matrix for the resource
};

struct Jobstr       // structure to hold the job details
{
    int prodtype;     // which product the job belongs to
    int arrival;      // arrival time of the job
    int duedate;      // due date of the job
 };

//------------------------------------------------------------ Function prototype declarations

int DP(float Fx[], int n, long *seed, int producttype[]);
//============================================================================== Main()
int main(int argc, char *argv[])
{
 char inpfile[25]="";            // stores the input name of the dat file
 char indata[75]="";             // stores the whole path of input dat file
 char outdata[75]="";            // stores the whole path for the output prb file
 char dpath[30]= "problem/";     // data path for the problem file input name
 char outpath[30] = "probinst/"; // data path for storing output prb file

 int jobs, products;                    // Number of jobs and number of products
 float setupsevfact;                    // Setup time severity factor
 float setupdeviation;              // Setup time deviation factor for resources
 float loadfactor;              // Shop Load factor (Rho)
 float phi;                     // grand mean of processing times
 float prdfactordev;                 // Deviation factor from phi for all products
 float procfactordev;        // Deviation factor for processes for individual process time
 float ops, loops;                      // stores number of operations and number of loops
 int capacity;                          // capacity - machines at each work centre
 float tardyfactor;             // percentage of the jobs tardy

 int i, j, k, l;     // counter variables

 float pmixtemp;                        // temporary variable
 int maxloops = 0;
 int maxops = 0;
 int resources;                         // Number of resources
 float rnum1;
 int rnum2;
 int operations, process;           // number of operations and processes for each product
 int m;                            // index variable for repeating the route for re-entrancy
 int lowprdlimit, upprdlimit;           // lower and upper limit for process time - products
 int lowproclim, upproclim;         // lower and upper limit for process time - processes
 int True1, True2, True3, True4;        // true/false - decision
 int proctimei;                  // stores individual processing time - Uniform Distribution
 int ptimeij;                   // sampled process time - Normal distribution
 int totaltime;                     // stores the sampled processing time across all processes
 float sigmapij;                    // mean processing time from samples
 float checkptime, samplephi;
 float setuprsrc;                   // stores the mean setup time for resources
 int lowsetuplimit, upsetuplimit;       // lower and upper limit of Unif Dist for setup time
```

```
int setupr;                    // sampled setup time for the resource
int setupprod;                 // sampled setup time for products
float samplesbar;                     // sample mean of setups
int lowprdsetup, upprdsetup;          // low and high limits of uniform distribution
int setupprdtemp;              // exchange variable
float sigmasrik;                // sum of setup times calculated
int Tnow;                       // variable to hold the current time
int TBA;                       // time between arrivals
int producttype[SIZE+1];
float lamda;                   // stores arrival rate
int ddate;                      // stores the derived due date
int tasks;                     // stores the number of tasks for simulation

long seed = 12345;                     // should be an input from the dat file

FILE *input, *output;   // file pointer for the input and output files

//------------------------------------------------------------------------ Initialize variables
pmixtemp = 0.0;
resources = 0;
totaltime = 0;
sigmapij = 0;
sigmasrik = 0;
Tnow = 0;
TBA = 0;
tasks = 0;

   //------------------------------------------------------------------------ Get Input file name
if(argc >1)  // section that reads the details the input file
{
   if(!strcmp(argv[1], "-?"))    // let the user know what is the expected format
   {
    cout <<"\n" <<"Usage: generator [problem data file name] {.dat extension is assumed}
      [seed] " <<"\n";
    exit (0);
   }
   else {
       strcpy(inpfile, argv[1]);  // copy the name of the input file from command line argument
       seed = atol( argv[2] );         // can set seed from command line
   }
}
else             // ask the user to input the file name
{
   cout <<"\n" <<"Input the problem data file name :";
   gets(inpfile);   // get the file name
}
sprintf(indata, "%s%s.dat", dpath,inpfile); // copy input file name and data path to indata

if((input=fopen(indata,"r"))==NULL)       // Open the input file in read mode and check
{
    printf("Input file opening failed.\n");   // Input file opening failed
    exit(-1);                       // exit from the program
}
else                                    // file opened successfully
    printf("\nInput file opened successfully. Reading in progress.....\n");

//------------------------------------------------------------------------ Read data from file

fscanf(input, "%d\n", &jobs);           // Read number of jobs from file
fscanf(input, "%d\n", &products);       // Read number of products from file
fscanf(input, "%f\n", &setupsevfact);   // Read the setup severity factor from file
fscanf(input, "%f\n", &setupdeviation); // Read setup deviation from mean in % from file

float *mixarray = new float[products+1];  // Dynamically allocate memory for mixarray
for(i=1;i<=products;i++)                  // How many % of jobs belong to each product
       fscanf(input, "%f", &mixarray[i]);     // Read values into mix array
```

```
 fscanf(input, "%f\n", &loadfactor);       // Read the load factor (Rho)
 fscanf(input, "%f\n", &phi);             // Grand mean of processing time for all products
 fscanf(input, "%f\n", &prdfactordev);  // Deviation factor across product type for process time
 fscanf(input, "%f\n", &procfactordev);    // Deviation factor across processes

 Productstr *productarray[products+1];     // Array of structure for products - dynamic allocation
 for(i=1;i<=products;i++)
 {
     productarray[i] = new Productstr;       // New instance of structure - dynamic allocation
     fscanf(input, "%f\n", &productarray[i]->psfactor);  // read values
 }

 fscanf(input, "%f %f \n", &ops, &loops);   // number of operations and loops from file
 fscanf(input, "%d \n", &capacity);         // capacity from file
 fscanf(input, "%f\n", &tardyfactor);     // reads the percentage of jobs expected to be tardy
 fclose(input);                             // close input file

//---------------------------------------- Print the read data - for debugging; remove at end
#if DETAIL
printf("No. of Jobs = %d \n", jobs);
printf("No. of Products = %d \n", products);
printf("Setup Severity Factor = %5.3f \n", setupsevfact);
printf("Setup Deviation Factor for Resources = %5.3f \n", setupdeviation);
printf("Mix Array Values = ");
for(k=1;k<=products;k++)
    printf("%5.3f  ", mixarray[k]);
printf("\n");
printf("Load Factor (Rho) = %5.3f \n", loadfactor);
printf("Grand Mean of Processing Times (Phi) = %5.3f \n", phi);
printf("Deviation Factor on Product for Process Time = %5.3f \n", prdfactordev);
printf("Deviation Factor on Processes for process time = %5.3f \n", procfactordev);
printf("Product Array :\n");
for(k=1;k<=products;k++)
    printf("%5.3f \n", productarray[k]->psfactor);
printf("Ops = %5.3f \n", ops);
printf("Loops = %5.3f \n", loops);
printf("Capacity = %d \n", capacity);
printf("Percentage of Tardy jobs = %5.3f \n", tardyfactor);
#endif
//------------------------------------------------------------------------- Open the output file
sprintf(outdata, "%s%s_%d%d%1.0f%d_%ld.prb", outpath,inpfile,jobs,products,loops,capacity,seed);
                    // copy output file name and data path to outdata
if((output=fopen(outdata,"w"))==NULL)       // Open the output file in write mode and check
 {
     printf("Output file opening failed.\n");   // Output file opening failed
     exit(-1);                                  // exit from the program
 }
 else                                        // file opened successfully
     printf("\nOutput file opened successfully. Writing in progress.....\n");

//------------------------------------------------------------------------- Calculate Data
 float *Fx = new float[products+1];         // Cumulative distribution array for products
 for(i=1;i<=products;i++)  //cumulative distribution for products
 {
     pmixtemp = pmixtemp + mixarray[i];
     Fx[i] = pmixtemp;        // Dynamic array Fx stores the cumulated product mix
 }

#if DETAIL
 printf("Fx Array = ");    // print the values of Fx array - debug help
 for(k=1;k<=products;k++)
     printf("%5.3f ", Fx[k]);
 printf("\n");
#endif

 for(i=1;i<=products;i++)  // calculate no. of operations, loops and resources for each product
 {
```

```
    productarray[i]->opsno = (int) ops;       // conversion of float to int
    productarray[i]->loopsno = (int) loops;   // conversion of float to int
    if(productarray[i]->opsno > resources)  // make no. of resources atleast equal to operations
           resources = productarray[i]->opsno;
    if(productarray[i]->loopsno > maxloops) // make max. loops atleast equal
                                            //to loops no.of a product
           maxloops = productarray[i]->loopsno;
    if(productarray[i]->opsno > maxops)   // make max. operations atleast equal to no.
                                          //of operations
           maxops = productarray[i]->opsno;
}

int *resourcearray = new int[resources+1];  // Dynamically allocate memory for resource array
for(i=1;i<=resources;i++)        // Create the resource array
       resourcearray[i] = capacity;  // assign the capacity for each resource

#if DETAIL
printf("Resource Array = ");    // print the values of resource array - debug help
for(k=1;k<=resources;k++)
    printf("%d ", resourcearray[k]);
printf("\n");
#endif

for(i=1;i<=products;i++)   // initialize all elements of sequence array of a product to -1
{
    for(j=1;j<=SIZE;j++)
        productarray[i]->seqarray[j] = -1;
}

for(i=1;i<=products;i++)    // generate the sequence for each product
{
    for(j=1;j<=productarray[i]->opsno;j++)   // each work cell to be visited
    {
        rnum1 = Rnd(&seed);                    // call random number generating function
        rnum2 = (int) (rnum1*1000);            // why multiplied by 1000 ????????
        if((rnum2 >= 0) && (rnum2 < resources))  // check whether the obtained rand num is in range
        {
            for(k=1;k<=productarray[i]->opsno;k++)  // enumerate all work cells visited previously
            {
                if(rnum2 == productarray[i]->seqarray[k])
                       rnum2 = -1;                  // previously visited, not again (by flow shop definition)
            }   // end of for loop (variable - k)
            if(rnum2 >= 0)                      // found an acceptable work cell number
            {
                   productarray[i]->seqarray[j] = rnum2;   // assign the next operation to be done
            }
            else
            {
                j--;    // couldn't find the next redo again
            } // end of if (inner)
           }  // end of if (outer)
            else
                j--;    // Random number out of range, redo again
    }  // end of for loop (varibale - j)
    operations = productarray[i]->opsno;     // number of operations for each product
    process = productarray[i]->opsno * productarray[i]->loopsno;  // calculate the no. of processes
    for(l=(operations+1);l<=process;l++)    // index started at 1, loop to copy the route
    {
        m = l-operations;
        productarray[i]->seqarray[l] = productarray[i]->seqarray[m];   // same route for reentrancy
    } // end of for loop (varaible - l)
}   // end of for loop (variable - i) or all product types

#if DETAIL
for(i=1;i<=products;i++)
{
    printf("Route Array for product %d \n",i);    // print the values of route array - debug help
```

```
     for(k=1;k<=process;k++)
            printf("%d ", productarray[i]->seqarray[k]);
     printf("\n");
 }  // end of for loop (variable - i)
#endif

 lowprdlimit = (int)(phi - (phi * prdfactordev));  // calculate lower limit of uniform distribution
 upprdlimit = (int)(phi + (phi * prdfactordev));   // calculate upper limit of uniform distribution
#if DETAIL
 printf("low limit on product = %d \n", lowprdlimit);
 printf("up limit on product = %d \n", upprdlimit);
#endif

 for(i=1;i<=products;i++)     // calculate total processing time for each product
 {
     True1 = 0;    // Initialize to zero before while
     while(True1 == 0)
     {
         proctimei = (int)(Rnd(&seed)*upprdlimit);   // make the time closer to the mean
         if((proctimei >= lowprdlimit) && (proctimei <= upprdlimit))
                True1 = 1;        // found the random number between suitable range
     } // end of while loop (variable - True1)
     productarray[i]->indpt = proctimei;         // sampled mean processing time for a product
#if DETAIL
        printf("Process time = %d \n", productarray[i]->indpt);
#endif
        lowproclim = (int) (proctimei - (proctimei * procfactordev));
                                    // calculate lower limit for U Dist
        upproclim = (int) (proctimei + (proctimei * procfactordev));
                                   // calculate upper limit for U Dist
#if DETAIL
        printf("low limit on process = %d \n", lowproclim);
        printf("up limit on process = %d \n", upproclim);
#endif
        for(j=1;j<=process;j++)   // do for each process across the particular product type
        {
            True2 = 0;
            while(True2 == 0)
            {
                ptimeij = (int)(Rnd(&seed)*upproclim);
                                // sampled processing time for product i and process j
                if((ptimeij >= lowproclim) && (ptimeij <= upproclim))
                          True2 = 1;        // found the random number between suitable range
            } // end of while loop (variable - True2)
            productarray[i]->ptimei[j] = ptimeij;   // store the time across process
            totaltime = totaltime + ptimeij;         // add up processing time for each process
        } // end of for loop (varaibel - j)
        productarray[i]->totptime = totaltime;  // store the total processing time for the product (i)
        productarray[i]->totstime = 0;           // initialize setup time total to zero
#if DETAIL
        printf("Total time = %d \n", productarray[i]->totptime);
        checkptime = totaltime / process;   // check what is the average of samples
        printf("Average of samples = %5.3f \n",checkptime);
#endif
        sigmapij = sigmapij + productarray[i]->totptime;   // summation of all pij values
        totaltime = 0;     // reinitalize the total time to zero
 }  // end of for loop (variable - i)
 samplephi = sigmapij / (process*products);     // calculate the phi from the sampled values
#if DETAIL
    printf("Phi from sampled times = %5.3f \n", samplephi);
#endif

    setuprsrc = setupsevfact * samplephi;
                              // setuptime mean for resources - fraction of samplephi
#if DETAIL
    printf("Mean Setup time for resource = %5.3f \n", setuprsrc);
#endif
```

```
    lowsetuplimit = (int)(setuprsrc - (setuprsrc * setupdeviation));
                                // calculate lower limit of unif dist
    upsetuplimit = (int)(setuprsrc + (setuprsrc * setupdeviation));
                                // calculate upper limit of unif dist
#if DETAIL
        printf("low limit on setup time = %d \n", lowsetuplimit);
        printf("up limit on setup time = %d \n", upsetuplimit);
#endif

    Setupstr *setuparray[resources+1];  // Array of structure for resources for Seq dep setup time
    for(i=1;i<=resources;i++)   // calculate the sequence dependent setup times
    {
     setuparray[i] = new Setupstr;       // New instance or structure - dynamic allocation
     True3 = 0;    // Initialize to zero before while
     while(True3 == 0)
     {
         setupr = (int)(Rnd(&seed)*upsetuplimit);   // make the time closer to the mean
         if((setupr >= lowsetuplimit) && (setupr <= upsetuplimit))
                True3 = 1;        // found the random number between suitable range
     } // end of while loop (variable - True3)
     setuparray[i]->indsetup = setupr;     // store the sampled setup time
#if DETAIL
        printf("Setup time (resource) = %d \n", setuparray[i]->indsetup);
#endif
     for(j=1;j<=products;j++)
     {
         for(k=1;k<=products;k++)
         {
             if(j==k)  // same products - no setup required
             {
                 setuparray[i]->sdsetuptime[j][k] = 0;
             }
             else
             {
#if DETAIL
                printf("prodarry %d ptime = %d \n", j, productarray[j]->indpt);
                printf("prodarry %d ptime = %d \n", k, productarray[k]->indpt);
#endif
                lowprdsetup = (int)(productarray[j]->indpt*productarray[j]->psfactor);
                upprdsetup = (int)(productarray[k]->indpt*productarray[k]->psfactor);
                if(lowprdsetup > upprdsetup)   // low limit can get high due to factor - check
                {
                     setupprdtemp = lowprdsetup;
                     lowprdsetup = upprdsetup;
                     upprdsetup = setupprdtemp;
                }  // completed the swap of the limits
                True4 = 0;
                while(True4 == 0)
                {
                    setupprod = (int)(Rnd(&seed)*upprdsetup);   // make the time closer to the mean
                    if((setupprod >= lowprdsetup) && (setupprod <= upprdsetup))
                        True4 = 1;        // found the random number between suitable range
                } // end of while loop (variable - True4)
#if DETAIL
                printf("setup time for product = %d \n", setupprod);
#endif
                setuparray[i]->sdsetuptime[j][k] = setuparray[i]->indsetup + setupprod;
                productarray[j]->totstime = productarray[j]->totstime + setuparray[i]->sdsetuptime[j][k];
                productarray[k]->totstime = productarray[k]->totstime + setuparray[i]->sdsetuptime[j][k];
                sigmasrik = sigmasrik + setuparray[i]->sdsetuptime[j][k];   // summation of setup times
             }  // end of if
         }  // end of for loop (variable - k)
     }  // end of for loop (variable - j)
#if DETAIL
            printf("Sigma srik = %5.3f \n", sigmasrik);
#endif
 }  // end of for loop (variable - i)
```

```
#if DETAIL
    for(i=1;i<=resources;i++)
    {
        printf("Resource = %d \n", i);
        for(j=1;j<=products;j++)
        {
            for(k=1;k<=products;k++)
                 printf(" %d", setuparray[i]->sdsetuptime[j][k]);
            printf("\n");
        }  // end of for (variable - j)
        printf("\n");
    }  // end of for (variable -i)
#endif
    samplesbar = sigmasrik / (resources * products);
#if DETAIL
    printf("Sample Setup time mean = %5.3f \n", samplesbar);
#endif

    int *jobprodnum = new int[jobs+1];
    for(i=1;i<=jobs;i++)   // decides in which order the jobs has to be released into the facility
        jobprodnum[i] = DP(Fx, products, &seed, producttype);

#if DETAIL
    printf("Job Product Number Array = ");    // print the values of jobprodnum - debug help
    for(k=1;k<=jobs;k++)
     printf("%d ", jobprodnum[k]);
    printf("\n");
#endif

// Calculate the arrival time, due date of the jobs that belong to each product type
    lamda = (loadfactor / (samplephi + samplesbar))*loops;
#if DETAIL
    printf("Value of Lamda calculated = %5.3f \n", lamda);
#endif
    Jobstr *jobarray[jobs+1];     // Array of structure for jobs
    for(i=1;i<=jobs;i++)
    {
         jobarray[i] = new Jobstr;     // New instance of structure - dynamic allocation
         jobarray[i]->prodtype = jobprodnum[i];  // store which product type the job belongs to
         jobarray[i]->arrival = Tnow;            // first job available at Tnow = 0
         for(j=1;j<=products;j++)
         {
             if(j == jobprodnum[i])          // calculate due date for the job
             {
#if DETAIL
                    printf("Total Setup time of product %d is %d \n", j, productarray[j]->totstime);
#endif
                    ddate = (int)((productarray[j]->totptime + productarray[j]->totstime)*(1 - tardyfactor));
                    jobarray[i]->duedate = jobarray[i]->arrival + ddate;
             }
         } // end of for loop (variable - j)
         TBA = ceil((expntl(lamda)));   // sample the time between arrival for next job
#if DETAIL
         printf("TBA = %d \n", TBA);
#endif
         Tnow = Tnow + TBA;            // Arrival time for the next job
     } // end of for loop (variable - i)
#if DETAIL
     for(i=1;i<=jobs;i++)
     {
         printf("Job number = %d \n", i);
         printf("Job product Type = %d ", jobarray[i]->prodtype);
         printf("Arrival Time = %d ", jobarray[i]->arrival);
         printf("Due Date = %d \n", jobarray[i]->duedate);
     }
#endif
```

```
// calculate the number of tasks associated with the simulation
    for(i=1;i<=jobs;i++)
        tasks = tasks + (productarray[jobprodnum[i]]->opsno*productarray[jobprodnum[i]]->loopsno);
#if DETAIL
    printf("Total No of tasks for simulation = %d \n", tasks);
#endif

//---------------------------------------------------------------------- Output to prb file
 fprintf(output, "%s\n", inpfile);        // print the file name
 fprintf(output, "%d %d %d\n", jobs, products, resources);    // write these values to file

 for(i=1;i<=resources;i++)
 {
     fprintf(output, "%d\n", resourcearray[i]);
     for(j=1;j<=products;j++)
     {
         for(k=1;k<=products;k++)
             fprintf(output, "%d ", setuparray[i]->sdsetuptime[j][k]);
         fprintf(output, "\n");;
     }  // end of for loop (variable - j)
 } // end of for loop (variable - i)

 fprintf(output, "%d\n", tasks);

 for(i=1;i<=jobs;i++)   // machine # and process time for each job
 {
     fprintf(output, "%d\n", jobprodnum[i]);
     process = productarray[jobprodnum[i]]->opsno * productarray[jobprodnum[i]]->loopsno;
     fprintf(output, "%d\n", process);
     for(j=1;j<=process;j++)
         fprintf(output, "%d %d ", productarray[jobprodnum[i]]->seqarray[j],
                                   productarray[jobprodnum[i]]->ptimei[j]);
     fprintf(output, "\n");
 } // end of for loop (variable - i)

 for(i=1;i<=jobs;i++)   //arrival time for each job
        fprintf(output, "%d\n", jobarray[i]->arrival);

 for(i=1;i<=jobs;i++)
        fprintf(output, "%d\n", jobarray[i]->duedate);

 fprintf(output, "%f %f\n", ops, loops);   // average operations and loops

 fprintf(output, "%f %f\n", setupsevfact, setupdeviation);

 fclose(output);    // writing completed, close output file

//---------------------------------------------------------------------- Delete the dynamic arrays
delete [] mixarray;           // Free the memory for the whole product mix array
delete [] resourcearray;      // Free the memory allocated for resource array
delete [] jobprodnum;         // Free the memory for the whole job product number array

 return (1);   // End of main function
}

/*============================================================================================ DP()
     DP function was originally written by Robin Kerbel
     It was used as the part of the first problem generator
*/
// Modified by - Deepu Philip; for C++
// Modified on : 04/14/2004

int DP(float Fx[], int n, long *seed, int producttype[])
{
 float r;
 int i;
 r = Rnd(seed);
```

```
 for (i=0;i<n;i++)
 {
        if (r <= Fx[i])
            {
            producttype[i] = producttype[i] + 1;
            return i;
        } // end of if
 }  // end of for loop (variable - i)
} // end of function DP()
```

The next section describes the utility functions used in problem generator. These external files are combined together with the help of a make file. The external files define the distributions used in the program, platform independent random number generation, function header declarations etc. They all are listed in the section below.

```
/*==========================================================================

     Sim++ version 1.01 is the property of Paul A. Fishwick and Robert M. Cubert,
     copyright (c) 1995; see license in the root directory of the distribution.

Distributions.cpp  ---> taken from capiran.cpp
----------------------------------------------------------------------------
simpack c-language application progrmamming interface
Part 3: pseudo-random-number-related.
----------------------------------------------------------------------------
These random number generator routines were written by M.H.  MacDougall,
"Simulating Computer Systems", MIT Press, 1987.

Code was extracted 950412 from queuing.c in "old simpack" into this file.  i
changed the function parameter and return declarations.  other than that i made
almost NO changes, for fear of altering the way the calculations work.

/// There are numerous error-checks in this code which have been commented out.
/// Do we want to re-enable them?  i'd guess yes.
----------------------------------------------------------------------------*/

//#include "queuing.h"
#include <math.h> // log(), sqrt()

#define A 16807L           /* multiplier (7**5) for 'ranf' */
#define M 2147483647L      /* modulus (2**31-1) for 'ranf' */
#define then

typedef double real;


long In0 [] =
    {
```

```
                          0L,   1973272912L,  747177549L,   20464843L,
           640830765L,   1098742207L,   78126602L,   84743774L,
           831312807L,    124667236L, 1172177002L, 1124933064L,
         1223960546L,   1878892440L, 1449793615L,  553303732L
    };

long In [] =
    {
                          0L,   1973272912L,  747177549L,   20464843L,
           640830765L,   1098742207L,   78126602L,   84743774L,
           831312807L,    124667236L, 1172177002L, 1124933064L,
         1223960546L,   1878892440L, 1449793615L,  553303732L
    };

int strm = 1;               /* index of current stream */
/*--------------------  SELECT GENERATOR STREAM  ---------------------*/
int stream (int n)
        { /* set stream for 1<=n<=15, return stream for n=0 */
            if ( (n<0) || (n>15) ); //ErrXit (9032, "bad n");
            if (n) strm=n;
            return(strm);
        }

void ResetRNstream () // internal use, NOT in API
    {
    int ii;
    for (ii = 0; ii < 16; ii++)
        In [ii] = In0 [ii];
    }//end fn



/*-------------  UNIFORM [0, 1] RANDOM NUMBER GENERATOR  -------------*/
/*                                                                     */
/* This implementation is for Intel 8086/8 and 80286/386 CPUs using   */
/* C compilers with 16-bit short integers and 32-bit long integers.   */
/*                                                                     */
/*---------------------------------------------------------------------*/

real ranf()
    {
        short *p,*q,k; long Hi,Lo;
        /* generate product using double precision simulation  (comments  */
        /* refer to In's lower 16 bits as "L", its upper 16 bits as "H")  */
        p=(short *)&In[strm]; Hi= *(p+1)*A;                 /* 16807*H->Hi */
        *(p+1)=0; Lo=In[strm]*A;                           /* 16807*L->Lo */
        p=(short *)&Lo; Hi+= *(p+1);    /* add high-order bits of Lo to Hi */
        q=(short *)&Hi;                       /* low-order bits of Hi->LO */
        *(p+1)= *q&0X7FFF;                                /* clear sign bit */
        k= *(q+1)<<1; if (*q&0X8000) then k++;         /* Hi bits 31-45->K */
        /* form Z + K [- M] (where Z=Lo): presubtract M to avoid overflow */
        Lo-=M; Lo+=k; if (Lo<0) then Lo+=M;
        In[strm]=Lo;
        return((real)Lo*4.656612875E-10);             /* Lo x 1/(2**31-1) */
    }



/*--------------  EXPONENTIAL RANDOM VARIATE GENERATOR  --------------*/
real expntl (real x)
        { /* 'expntl' returns a psuedo-random variate from a negative     */
            /* exponential distribution with mean x.                        */
            return(-x*log(ranf()));
        }

/*----------------  ERLANG RANDOM VARIATE GENERATOR  -----------------*/
```

```
real erlang (real x, real s)
        { /* 'erlang' returns a psuedo-random variate from an erlang      */
            /* distribution with mean x and standard deviation s.           */
            int i,k; real z;
            /* if (s>x) then error(0,"erlang Argument Error: s > x"); */
            z=x/s;
            k = int (z*z);
            z=1.0; for (i=0; i<k; i++) z *= ranf();
            return(-(x/k)*log(z));
        }



/*-----------------  NORMAL RANDOM VARIATE GENERATOR  ----------------*/
real normal (real x, real s)
        { /* 'normal' returns a psuedo-random variate from a normal dis-  */
            /* tribution with mean x and standard deviation s.              */
            real v1,v2,w,z1; static real z2=0.0;
            if (z2!=0.0)
                then {z1=z2; z2=0.0;}  /* use value from previous call */
                else
                    {
                        do
                            {v1=2.0*ranf()-1.0; v2=2.0*ranf()-1.0; w=v1*v1+v2*v2;}
                        while (w>=1.0);
                        w=sqrt((-2.0*log(w))/w); z1=v1*w; z2=v2*w;
                    }
            return(x+z1*s);
    }
//------------------------------------------------------------- Poisson()
//
// PURPOSE: Generates poisson distributed random numbers
//------------------------------------------------------------------------
/*
Note: Formula derived from Law & Kelton.
            1. Let a = e ^-y, b = 1, and i = 0;
            2. Generate U(i+1) ~ U( 0, 1 ) and replace b by b*U(i+1).
                 If b < a, return X = 1. Otherwise, go to step 3.
            3. Replace i by i+1 and go back to step 2. [ Law & Kelton, 1991 ]

Parameters: The mean lamda is a positive real number.

Range: [ 0, 1, 2, ..., ]

Mean: lamda
Var:  lamda

"The Poisson distribution is a discrete distribution that is often
used to model the number of random events occurring in an interval
of time. If the time between events is exponentially distributed,
then the number of events that occur in a fixed time interval has a
Poisson distribution. The Poisson distribution is also used to model
random variations in batch size." [ Pegden et al, 1990 ]

Reference: Introduction to Simulation using SIMAN by C. Dennis Pegden,
                    Robert E. Shannon, and Randall P. Sadowski, 1990., pg., 565.

                    Simulation Modeling & Analysis by Averill M. Law and W. David
                    Kelton, 1991, pg., 503.

Adapted from Joseph A Fisher randnum library ... 3/25/02 elm
------------------------------------------------------------------------
*/
long Poisson ( float lamda, long *seed  )   {
    double b = 1.0, a;
    long count = 0;
```

```
    a = expntl( lamda );

    do   {
        b = b * Rnd( seed );
        count++;
    } while ( b >= a );

     return count-1;

}
// end Distributions.cpp


/* --------------------------- rutil.h -------------------------------- */

/* Definitions */

    float Rnd( long *seed );          /* return U[0,1] val */
    float Rnorm( short *iseven, float *enorm, long *seed );
                                                          /* return Normal(mu,sigma) */



/*-------------------------------- rutil.c ------------------------------ */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

/* ------------------------------------------------------------------ Rnd()
    uniform (0,1) random number generator multiplicative congruential
            method:  z(i)=(7^5*z(i-1))(mod 2^31 - 1)

            from : law & kelton, simulation modeling and analysis,
                    mcgrawhill, 1982, p. 227.

*/
float Rnd( long *seed )
{
 /*  seed = seed*a mod p  */
    static long a = { 16807l };       /* 7^5 */
    static long b15 = { 32768l };     /* 2^15 */
    static long b16 = { 65536l };     /* 2^16 */
    static long p = { 2147483647l };  /* 2^31 - 1 */
    long xhi,xalo,leftlo,fhi,k;
    float r;

    xhi = *seed/b16;           /* Get 15 Hi Order Bits of seed */
    xalo = (*seed-xhi*b16)*a;  /* Get 16 lo bits of seed & form lo product */
    leftlo = xalo/b16;         /* Get 15 hi order bits of lo product */
    fhi = xhi*a+leftlo;        /* Form the 31 highest bits of full product */
    k = fhi/b15;               /* Get overflow past 31st bit of full product */
    *seed = (((xalo-leftlo*b16)-p)+(fhi-k*b15)*b16)+k;
    if ( *seed < 0 ) *seed = *seed+p;   /* Add back p if necessary */

/*$$ printf ("Z = %ld  ", *seed); */

    r = ((float) *seed) * ((float) 4.656612875e-10); /* Divide by (2^31-1) */
/*  fprintf(outfile, "** In Rnd() r = %.5f\n", r ); */
    return r;

} /* end Rnd() */


/* ------------------------------------------------------------------ Rnorm()
    Normal deviate generator ... generates N(0,1) samples in pairs
```

```
    iseven is set to zero in the calling program and maintained by RNORM
         to indicate whether a new pair of deviates needs to be generated
    enorm is returned on even numbered calls

    from : Pritsker and Pegden, "Introduction to simulation with SLAM,
            John Wiley and Sons, 1979, appendix G.

*/
float Rnorm( short *iseven, float *enorm, long *seed )
{
 float ua, ub, w, x;

    if ( ! *iseven ) {
        do {
            ua = 2.*Rnd( seed ) - 1.0;
            ub = 2.*Rnd( seed ) - 1.0;
            w = ua*ua + ub*ub;
        } while ( w > 1.0 );
        w = sqrt( (double) -2.*log( w )/w );
        *enorm = ub*w;
        *iseven = 1;
        x = ua*w;
    }
    else {
        *iseven = 0;
        x = *enorm;
    }

    return x;

} /* end Rnorm() */


/* --------------------------- util.h ---------------------------------- */

#define DOS         1   /* Microsoft and most DOS C compilers */
#define VM_SASC     2   /* SAS C compiler for IBM 3090 VM */
#define UNIX_AIX    3   /* cc compiler for IBM 6000 AIX */
#define UNIX_OSF    4   /* cc compiler for DEC OSF */
#define GCC         5   /* gnu cc compiler for linux */

#define OPSYS GCC


/*--------------------------------- Util.c ------------------------------- */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <time.h>


#include "util.h"
#if OPSYS == GCC
#include <unistd.h>    // for linux getcwd()
#else
#include <direct.h>
#endif


# Linux make file for the problem generator
PROJ=generator
DEBUG=
CC=g++
LIBDIR = .
CFLAGS= -g -ansi -I.
```

```
LDFLAGS= -lm
OBJS_EXT = util.o
LIBS_EXT =
#
# default inference rule
# c.o:
#   $(CC) $(CFLAGS) -c $<
#

$(PROJ):    $(PROJ).o $(OBJS_EXT) $(LIBS_EXT)
        $(CC) $(PROJ).o $(OBJS_EXT) $(LIBS_EXT) -o $(PROJ) $(LDFLAGS)

$(PROJ).o:  $(PROJ).cpp
        $(CC) $(CFLAGS) -c $(PROJ).cpp

$(OBJS_EXT).o:  $(OBJS_EXT).cpp
        $(CC) $(CFLAGS) -c $(OBJS_EXT).cpp
```

A sample input file is shown below. The input file has the parameters listed in Table 31.

```
10
2
0.2
0.1
0.5
0.5
0.3
20
0.2
0.1
0.1
0.2
2
2
1
0.3
```

The sample problem output from generator is summarized below.

```
4
10 4 3
3
0.000000 6.984509 21.085947 11.241922
0.000000 0.000000 0.000000 0.000000
29.022707 15.927137 0.000000 -5.664434
47.153111 37.403557 13.717620 0.000000
3
```

```
0.000000 3.349200 10.115778 15.399632
0.000000 0.000000 0.000000 0.000000
10.764121 8.970617 0.000000 13.206804
23.284748 16.900990 33.845722 0.000000
3
0.000000 15.283535 12.151340 2.644336
0.000000 0.000000 0.000000 0.000000
14.018339 12.945758 0.000000 6.083713
16.070204 28.796629 -12.685073 0.000000
90
3
9
2 51  1 56  0 33  2 31  1 42  0 53  2 45  1 25  0 37
1
9
2 0  1 2  0 2  2 2  1 0  0 0  2 0  1 2  0 1
3
9
2 51  1 56  0 33  2 31  1 42  0 53  2 45  1 25  0 37
1
9
2 0  1 2  0 2  2 2  1 0  0 0  2 0  1 2  0 1
3
9
2 51  1 56  0 33  2 31  1 42  0 53  2 45  1 25  0 37
3
9
2 51  1 56  0 33  2 31  1 42  0 53  2 45  1 25  0 37
1
9
2 0  1 2  0 2  2 2  1 0  0 0  2 0  1 2  0 1
3
9
2 51  1 56  0 33  2 31  1 42  0 53  2 45  1 25  0 37
2
9
2 2  0 2  1 2  2 0  0 2  1 0  2 1  0 2  1 1
3
9
2 51  1 56  0 33  2 31  1 42  0 53  2 45  1 25  0 37
87
93
392
405
492
502
502
578
586
715
7547
453
7852
765
7952
7962
862
8038
610
8175
3.000000 3.000000
3.000000 1.000000
```

APPENDIX B - SIMULATION

The simulation mode obtained the necessary input values for various parameters from a .prb file. The input to the simulator followed the order given in Table 32 below.

Table 32: List of input parameters for the simulator

Sl. No	Field Name	Data Type	Description
1	njob	Integer	Stores the number of jobs
2	nproduct	Integer	Stores the number of product types
3	setuptime	Float	Value of sequence dependent setup times
5	ntask	Float	Number of tasks that belong to each product
6	loadfactor	Float	How heavily the shop is loaded (Rho)
7	phi	Float	Grand mean of process times for the shop
8	prdfactordev	Float	Deviation from phi for products
9	procfactordev	Float	Deviation for processing time - processes
10	psfactor	Float	Product setup time as % of its process time
11	ops	Float	Number of operations
12	loops	Float	No of loops (reentrant loop number)
13	capacity	Integer	Capacity at each work cell
14	tardyfactor	Float	% of jobs that are tardy
15	probname	String	name of the problem
16	nres	Float	Number of resource available
17	nunit	Float	Number of unit of each resource (capacity)

Now the source code used in the program is listed below. First we list the objects used in the simulation, followed by the rest of the program.

```
// DLList.h: interface for the DLList class.
//
//////////////////////////////////////////////////////////////////////

#ifndef _DLLIST_H
#define _DLLIST_H 1

struct elem
{
    void *info; //pointer points to the informaiton
    elem *prev; //pointer points to the previous element
    elem *next; //pointer points to the next element
};

class DLList
{
public:
    DLList();               //constructor
```

```
    virtual ~DLList();      //destructor
    void Start();           //start
    elem *Forward();        //forward
    elem *Backward();       //backward
    void Append(void *);    //append to the tail
    void Insert(void *,int (*comp) (void*, void*)); //insert into the list
    elem *FindElem(void *); //find the element.
    void RmElem(elem *);    //remove an element from the list
    int Get_ItemCount();    //get the total number of items
      elem *Get_Curr();       // get current pointer
private:
    elem *pStart;           //pointer points to the start of the list
    elem *pCur;             //pointer points to the current point of the list
    int ItemCount;          //the number of items in the list

};

#endif


// Event Structure definition Calendar object prototypes
#ifndef _CALENDAR_H
#define _CALENDAR_H 1

struct Event  {    // Define a row (record) in the event calendar
  char Etype;        // Event type - A(rrival), B(egin) or E(nd) service
  float Time;        // Event time
  void *Einfo;       // Information needed by event routines (e.g. structure)
};

class Calendar {
public:
   Calendar(int size, int increment);  // constructor
   virtual ~Calendar();     // Destructor
   void Schedule( char Etype, float ETime, void *Einfo ); // Schedule an event
   Event remove();
   int empty(){return n == 0;}
   void display();
protected:
   Event *a;
   int N, n, ChunkSize;
   int parent(int i){return (i-1)/2;}
   int left(int i){return 2 * i + 1;}
   int right(int i){return 2 * i + 2;}
   void insert(Event x);
   void sift0();
};
#endif

// Statistical Accumulator Structures & Functions for Discrete Change Variables
#ifndef _DSTAT_H
#define _DSTAT_H 1

struct DSTAT {
  float Sum;           // Sum
  float SumSq;         // Sum squared
  float Min;
  float Max;
  float SumDt;         // Sum of delta t's
  float Tlast;         // last time updated
  char Title[100];     // Title for output
};

class Dstat {    // Discrete Change Accumulator Class
public:
   Dstat( char *title );     // constructor
   virtual ~Dstat();                    // Destructor
```

```
   void SumDstat( float value );
   void PrntDstat( float value );
protected:
   DSTAT *Accum;
};

#endif

// Statistical Accumulator Structures & Functions for Observation-based (TALLY) Variables
#ifndef _TALLY_H
#define _TALLY_H 1

struct TALLY {
  float Sum;           // Sum
  float SumSq;         // Sum squared
  float Min;
  float Max;
  long Nobs;          // Number of observations
  char Title[100];     // Title for output
};

class Tally {    // Discrete Change Accumulator Class
public:
  Tally( char *title );                 // constructor
  ~Tally();                             // Destructor
  void SumTally( float value );
  void PrntTally();
protected:
   TALLY *Accum;
};

#endif

// -------------------------- general.h -------------------------------
// Misc. constants, etc. used in multiple files.
#ifndef _GENERAL_H
#define _GENERAL_H 1


#define TRUE 1
#define FALSE 0
#define INFINITY 100000     // a big number

#define TRACE FALSE        // Trace output toggle
#define DEBUG FALSE         // Debug output toggle

#endif

// ----------------   Calendar.cpp -------------------------
// Event calendar (LVF priority queue) implemented as a heap
//
// Adapted from priorq.h template from Chapter 8 of
//   Ammeraal, L. (1996) Algorithms and Data Structures
//   in C++, Chichester: John Wiley.
// elm 1/5/2001

#include "calendar.h"
#include <iostream.h>
#include <iomanip.h>

#define nil (-1)

//Calendar(int size, int increment=10)  //Constructor
// size = initial size
// increment = chunk size for allocating memory

Calendar::Calendar(int size, int increment) {  // constructor
```

```
   N = size;
   a = new Event[N];    // get a pointer to block of events
   n = 0; ChunkSize = increment;
}

Calendar::~Calendar(){delete[]a;}      // Destructor

void Calendar::insert(Event x)
{  int i, j;

   //cout << "****Insert****" << endl;
   //display();
   if (n == N)
   {  Event *aOld = a;
      a = new Event[N += ChunkSize];
      for (i=0; i<n; i++) a[i] = aOld[i];
      delete[] aOld;
   }
   i = n++;
   while (i > 0 && x.Time < a[j = parent(i)].Time) {
     a[i] = a[j]; i = j;
   }
   a[i] = x;
//    cout << "****" << endl;
//    display();
}

Event Calendar::remove()
{  Event x = a[0];
//     cout << "****Remove****" << endl;
//     display();
   a[0] = a[--n];
   sift0();
//     cout << "****" << endl;
//     display();
   return x;
}

void Calendar::sift0()
{  int i = 0, j;
   Event x;
   x = a[0];
   while ((j = 2 * i + 1) < n)
   {  if (j < n - 1 && a[j+1].Time < a[j].Time) j++;
      if (a[j].Time < x.Time){a[i] = a[j]; i = j;} else break;
   }
   a[i] = x;
}

void Calendar::display()   // output Calendar contents for debug
{  int i;
   for (i=0; i<n; i++) {
     cout << "   a[i] ->:";
     cout << setw(3) << a[i].Etype << a[i].Time;
     cout << endl;
   }
}

//=================================================================== Schedule()
//  Schedules an event by inserting values into the calendar

void Calendar::Schedule( char Etype, float ETime, void *Einfo ) {
  Event Ebuf;

  Ebuf.Etype = Etype;
  Ebuf.Einfo = Einfo;
  Ebuf.Time = ETime;
```

```
  insert( Ebuf );

} // end Schedule()


// DLList.cpp: implementation of the DLList class.
//
//////////////////////////////////////////////////////////////////////

#include <stdlib.h>
#include "dllist.h"
#include <iostream.h>


//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

//constructor
DLList::DLList()
{
    pStart = new elem;  //create sentinel
    pStart->prev = pStart->next = pStart;   //only one elem
    pStart->info = NULL;
    pCur = NULL;
    ItemCount = 0;
}


//destructor
DLList::~DLList()
{

    elem *p,*pn;
    Start();
    p = pCur->next;
    for (;;)
    {
        if ( p == pStart ) break;   //at beginning ?
        pn = p->next;
        delete p;
        p = pn;
    }
    delete pStart;  //Sentinel

}

//member function: Start()
void DLList::Start()
{
    pCur = pStart;
}

//member function: Forward()
elem* DLList::Forward()
{
    pCur = pCur->next;
    return (pCur == pStart) ? NULL:pCur;
}

//member function: Backward()
elem* DLList::Backward()
{
    pCur = pCur->prev;
    return (pCur == pStart) ? NULL:pCur;
}
```

```
//member function: Append()
void DLList::Append(void *info)
{
    elem *p = new elem;
    p->info = info;
    elem *pEnd = pStart->prev;
    pEnd->next = p;
    p->prev = pEnd;
    p->next = pStart;
    pStart->prev = p;

    ItemCount = ItemCount +1;   //number of items increased by 1
}

//member function: Insert()
void DLList::Insert(void *info, int (*comp) (void*, void*))
{
    elem *ne, *p, *pl, *pn;
    void *ip, *ipl; //pointers to info
    ne = new elem;

    ne->info = info;

    //traverse the list to find the insertion point
    Start();

    pl = pCur;
    ipl = pl->info;
    while ( (p = Forward() ) != NULL)   //at beginning ?
    {
        ip = p->info;
        if ( (*comp) (info, ip) < 0 )   //should info precede ip ?
            if ( ipl == NULL )
                break;
            else if ( (*comp) (ipl,info) <= 0)
                break;  //found location
            pl = p;
            ipl = ip;
    }

    if ( p == NULL )    //place at the end of the list
    {
        pn = pStart;
        pl = pn->prev;
    }
    else
        pn = p;

    pl->next = ne;
    ne->prev = pl;
    ne->next = pn;
    pn->prev = ne;

    ItemCount = ItemCount +1;   //number of items increased by 1

}   //end of Insert()

elem* DLList::FindElem(void *info)  //find the element which must be transferred to void *
{
    elem *p;
    Start();
    do
        p = Forward();
    while ( p && (p->info != info) );

    return p;
```

```
}

//member function: RmElem(elem *p)
void DLList::RmElem(elem *p)    //remove a element from the list
{
    p->prev->next = p->next;
    p->next->prev = p->prev;
    delete p;
    ItemCount--;
}

//member function: Get_ItemCount()
int DLList::Get_ItemCount()
{
    return ItemCount;
}

elem* DLList:: Get_Curr()
{
    return pCur;
}


// ----------------   Dstat.cpp -------------------------
// Discrete Change variable Statistics Collection and display
//
#include <general.h>     // General constants, etc.
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "dstat.h"
extern float Tnow;                      // Current simulated time

//========================================================================== InitDstat()
//  Initializes Discrete Change Statistics (DSTAT) accumulators

Dstat::Dstat( char *title ) {           //Constructor
  Accum = new DSTAT;
  Accum->Sum = Accum->SumSq = 0;
  Accum->Min = INFINITY;
  Accum->Max = -INFINITY;
  Accum->SumDt = 0.0;
  Accum->Tlast = Tnow;    // sum of delta t's, last time updated
  strcpy(Accum->Title, title );
} // end Dstat()

Dstat::~Dstat(){delete Accum;}      // Destructor

//============================================================== SumDstat()
//  Updates Discrete Change Statistics (DSTAT) accumulators

void Dstat::SumDstat( float value ) {
  float dt;

  dt = Tnow - Accum->Tlast;

  Accum->Sum += value*dt;
  Accum->SumSq += value*value*dt;
  if ( Accum->Min > value ) Accum->Min = value;
  if ( Accum->Max < value ) Accum->Max = value;
  Accum->SumDt += dt;
  Accum->Tlast = Tnow;    // sum of delta t's, last time updated
} // end SumDstat()

//============================================================== PrntDstat()
//  Prints Summary Statistics for
```

```
//     Discrete Change Variables (DSTAT) accumulators

void Dstat::PrntDstat( float value ) {
  float s2;
  SumDstat( value );   // Collect statistics to here

  printf( "\n  %s", Accum->Title );
  printf( "\n    Current Value: %f", value );

  printf( "\n    Mean (xbar): %f", Accum->Sum/Accum->SumDt );

  s2 = ( Accum->SumDt*Accum->SumSq - Accum->Sum*Accum->Sum ) /
            ( Accum->SumDt*Accum->SumDt );     // Sample Variance
  printf( "\n    Variance (s^2): %f", s2 );
  printf( "\n    Maximum : %f", Accum->Max );
  printf( "\n    Minimum : %f", Accum->Min );
  printf( "\n    Collection Time: %.2f", Accum->SumDt );
  printf( "\n" );
} // end PrntDstat()

// ----------------   Tally.cpp -------------------------
// Observation-based variable Statistics Collection and display
//
#include <general.h>     // General constants, etc.
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "tally.h"

//=========================================================================== Tally()
//  Initializes Tally (observation) Statistics (TALLY) accumulators

Tally::Tally( char *title ) {           //Constructor
  Accum = new TALLY;
  Accum->Sum = Accum->SumSq = 0;
  Accum->Min = INFINITY;
  Accum->Max = -INFINITY;
  Accum->Nobs = 0;
  strcpy(Accum->Title, title );
} // end Tally()

Tally::~Tally(){delete Accum;}          // Destructor

//=========================================================================== SumTally()
//  Updates Tally (observation) Statistics (TALLY) accumulators

void Tally::SumTally( float value ) {
  Accum->Sum += value;
  Accum->SumSq += value*value;
  if ( Accum->Min > value ) Accum->Min = value;
  if ( Accum->Max < value ) Accum->Max = value;
  Accum->Nobs++;

} // end SumTally()

//=========================================================================== PrntTally()
//  Prints Summary Statistics for
//     Observation-based Variables (TALLY) accumulators

void Tally::PrntTally() {
  float s2;
  printf( "\n\n  %s",Accum->Title );
      printf( "\n    Mean (xbar): %f", Accum->Sum/Accum->Nobs );

      s2 =  ( Accum->Nobs*Accum->SumSq - Accum->Sum*Accum->Sum ) /
              (Accum->Nobs*(Accum->Nobs - 1) ) ;     // Sample Variance
```

```
      printf( "\n    Variance (s^2): %f", s2 );
      printf( "\n    Maximum : %f", Accum->Max );
      printf( "\n    Minimum : %f", Accum->Min );
      printf( "\n    Number of Observations : %d", Accum->Nobs );

} // end PrntTally()

// SIMPL Job & Task interface
#ifndef _JOB_H
#define _JOB_H 1

#include <dllist.h>

class Task {                    /** Task **/
public:
  Task( int number );     // constructor
  ~Task();                      // Destructor
  short num;        /* Task index */
  float dur;        /* task time */
  short rsrc;       /* Resource index */
  short job;        /* Job index */
  short jobstep;    /* Job step index */
  DLList *succlst;  /* Job precedence task successor adjacency list */
  DLList *predlst;  /* Job precedence task predecessor adjacency list */
  float TStart;   /* Task start time */
  float TEnd;     /* Task end time */
  float MTTS;     /* Mean time to ship */
  float RemTime;      /* Remaining time before shipping */
  int numsteps;   /* number of remaining steps */
};

class Job {                   /** Job **/
public:
  Job( int number );     // Constructor
  ~Job();                 // Destructor
  int Num;                // Job number
  float Atime;            // System Arrival Time
  Task *CurrTask;         // Current task in job task list
  int Prodnum;        // Product number
  int jobntasks;          // Number of tasks
  Task *FirstTask;    // First task in job task list
  float DueDate;          // Job Due Date
  float JEndTime;         // Job end time
  float AveSetup[10];     // Average setup array (for each machine)
};

#endif

// ----------------   Job.cpp --------------------------
// Jobs and task classes
//
#include <general.h>     // General constants, etc.
#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>    // C++ I/O
#include <string.h>

#include "job.h"
extern float Tnow;                      // Current simulated time
int m;

Job::Job( int number ) {   //Constructor
  Num = number;
  Atime = 0.0;
  CurrTask = NULL;
  Prodnum = -1;
```

```
  jobntasks = 0;
  FirstTask = NULL;
  DueDate = 0.0;
  JEndTime = -1.0;
  for (m=0;m<10;m++){
    AveSetup[m] = 0.0;
  }
}

Job::~Job(){}    // Destructor

Task::Task( int number ) {   //Constructor
  num = number;
  dur = 0.0;
  rsrc = job = jobstep = 0;
  succlst = new DLList;
  predlst = new DLList;
  TStart = -1.0;
  TEnd = -1.0;
  MTTS = 0.0;
  RemTime = 0.0;
  numsteps = 0;
}

Task::~Task(){delete succlst; delete predlst;}                  // Destructor


// SIMPL Queue interface
#ifndef _QUEUE_H
#define _QUEUE_H 1

// ** Queueing Discipline Constants
#define PRIORITY 1   // STATIC priority queue (need comparison function)
#define FIFO 2
#define MAXTSYS 3    // maximum time in system (dynamic priority)
#define CRRATIO 4    // lowest critical ratio
#define LSLACK 5       // lowest slack value
#define RANDOM 6     // random rule
#define SEARCH 7     // search mode operation

#include <dllist.h>
#include "job.h"
#include "dstat.h"

class Queue {    // General Queue object ... uses DLList's
public:
  int (Queue::*CmpJob) ( Job *job1 , Job *job2 );  // ptr. to function in class

  Queue( char *name, long capacity, int discipline,
              int (*comp) (void*, void*) );        // constructor
  ~Queue();                                             // Destructor
  void InsertQ( Job *item );
  Job *RetrieveQ( int tsknum );
  int NQ( void );
  void PrntNQStats( void );  // Print Dstats for NQ
  int InQueue( int tsknum ); // check to see whether the task ins in queue

  char Name[20];    // Name of the queue (for output)
  long Capacity;    // Queue Capacity ( IGNORED FOR NOW - IMPLEMENT LATER)
  int Discipline;   // Queuing discipline (LVF, HVF, FIFO, etc.)
  int (* cmpfunc) (void*, void* );  // pointer to comparison function (LVF, HVF)
  DLList *Qlist;    // List of entities in the queue.
  Dstat *NQAccum;   // Queue statistics array
};

#endif
```

```
// ----------------   Queue.cpp -------------------------
// Queues consist of lists of entities, in general
// Added queuing disciplines to the object
//
#include <general.h>     // General constants, etc.
#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>    // C++ I/O
#include <string.h>

#include "queue.h"
extern float Tnow;                      // Current simulated time

Queue::Queue( char *name, long capacity, int discipline,
              int (*comp) (void*, void*) ) {   //Constructor
  strcpy(Name, name );
  Capacity = capacity;
  Discipline = discipline;
  cmpfunc = *comp;
  Qlist = new DLList;
  NQAccum = new Dstat( strcat( name, "_NQ" ) );
}

Queue::~Queue(){delete Qlist; delete NQAccum;}                 // Destructor

//======================================================================== InsertQ()
// Place an job on the queue list BY Priority
// Returns 1 on success, 0 if the queue is full

void Queue::InsertQ( Job *item )
{
  NQAccum->SumDstat( (float) NQ() );  // Collect Nq statistics

  switch( Discipline ) {

  case PRIORITY :  {       // STATIC priority queues (need comparison function)
    Qlist->Insert( (void *) item, cmpfunc );
    }
    break;

  default: {                // Assume non-priority (FIFO, dynamic priority) default
    Qlist->Append( (void *) item );
    }
    break;
  } // end switch

   return;
} // end InsertQ()


//======================================================================== RetrieveQ()
//  Retrieve (a pointer to) an job from the queue list according LVF
//    discipline
//  Return pointer to first item on success, NULL if the queue is empty

Job *Queue::RetrieveQ( int tsknum )
{
  elem *p;
  Job *jobp;
  Task *taskp;

  int ctnum;

  ctnum = tsknum;  // assigned task number stored

  Qlist->Start();         // position at beginning of list
  if ( (p = Qlist->Forward() ) == NULL) {   //empty ?
```

```
  cout << "** WARNING - RetrieveQ() - No elements on list **";
}

NQAccum->SumDstat( (float) NQ() );  // Collect Nq statistics

switch( Discipline ) {

case MAXTSYS : {         // Traverse queue - find the minimum TISYS entry
    float tisys, maxtisys = -1.0e+20;
    elem *maxp = NULL;      // pointer to job with max time in sys.

    Qlist->Start();           // position at beginning of list
    while ( (p = Qlist->Forward() ) != NULL ) {
      jobp = (Job *) p->info;
      tisys = Tnow - jobp->Atime;
      if ( tisys > maxtisys ) {
        maxtisys = tisys;
        maxp = p;
      }
    } // end traverse
    jobp = (Job *) maxp->info;
    Qlist->RmElem( maxp );
  }
  break;

  case CRRATIO : {         // Traverse queue - find the minimum CR entry
    float due, cr, lowcr = 1.0e+20;
    int res, numsteps;
    float setup;
    elem *lowcrp = NULL;      // pointer to job with lowest CR.

    Qlist->Start();           // position at beginning of list
    while ( (p = Qlist->Forward() ) != NULL ) {
      jobp = (Job *) p->info;
      taskp = jobp->CurrTask;
    res = taskp->rsrc;
    setup = jobp->AveSetup[res];
      numsteps = taskp->numsteps;
      //printf("QUEUE: Product %d Resource %d Ave Setup %f\n", jobp->Prodnum, taskp->rsrc, setup);
    due = jobp->DueDate - Tnow;
    cr = due/(taskp->MTTS + (setup*numsteps));
      //printf("  QUEUE: Job %d Task %d CR Value %.2f\n", jobp->Num, taskp->num, cr);

      if ( cr <= lowcr ) {
        lowcr = cr;
        lowcrp = p;
      }
    } // end traverse

    jobp = (Job *) lowcrp->info;
    //printf("      SELECTED: Job %d Task %d CR Value %.2f\n",jobp->Num, taskp->num, lowcr);


    Qlist->RmElem( lowcrp );
  }
  break;

  case LSLACK : {         // Traverse queue - find the minimum slack entry
    float slack, lowslack = 1.0e+20;
    elem *lowslackp = NULL;      // pointer to job with lowest CR.

    Qlist->Start();           // position at beginning of list
    while ( (p = Qlist->Forward() ) != NULL ) {
      jobp = (Job *) p->info;
      taskp = jobp->CurrTask;
    slack = jobp->DueDate - Tnow - taskp->RemTime;
      //printf("QUEUE: Job %d Task %d Slack Value %.2f\n", jobp->Num, taskp->num, slack);
```

```
      if ( slack <= lowslack ) {
        lowslack = slack;
        lowslackp = p;
      }
     } // end traverse

     jobp = (Job *) lowslackp->info;
     //printf("    SELECTED: Job %d Task %d Slack Value %.2f\n",jobp->Num, taskp->num, lowslack);

     Qlist->RmElem( lowslackp );
   }
   break;

   case RANDOM : {
    elem *foundp = NULL;      // pointer to job in selected position.

    float rnum;     // for the generated radnsom number
    float scalernum;  // random number scaled for the application
    int randnum;     // random number converted to integer

    int maxnq, count;

    Qlist->Start();         // position at beginning of list
    if ( (p = Qlist->Forward() ) == NULL) {   //empty ?
       cout << "** WARNING - RetrieveQ() - No elements on list **";
    }

    NQAccum->SumDstat( (float) NQ() );  // Collect Nq statistics

    maxnq = 0;
    count = 0;
    Qlist->Start();           // position at beginning of list
    while ( (p = Qlist->Forward() ) != NULL ) {
     count = count + 1;
    }
    maxnq = count;

 if( TRACE ) printf("Number in queue = %d \n", maxnq);

 rnum = (float) rand() / RAND_MAX;   // generated random number
 scalernum = rnum * (maxnq+1);      // scale the generated random number
 randnum = (int) scalernum;          // convert to integer for selecting the job
 if(randnum == 0)
    randnum++;      // no zero job - hence convert into position 1

 count = 0;
 Qlist->Start();           // position at beginning of list
 while ( (p = Qlist->Forward() ) != NULL ) {
       count = count + 1;
       if (count == randnum)
       {
         foundp = p;
       }
  } // end traverse

  jobp = (Job *) foundp->info;
  taskp = jobp->CurrTask;
 // printf("QUEUE: Job %d Task %d Position Value %d\n", jobp->Num, taskp->num, count);

 Qlist->RmElem( foundp );
}
break;

case SEARCH : {               // when simulation is working as a search, dp, 07/17/2003
    elem *reqelm = NULL;
    int found = 0;
```

```
     int nq = 0;

     if( TRACE ) printf(" requested task : %d \n", ctnum);
     Qlist->Start();         // position at beginning of list
     if ( (p = Qlist->Forward() ) == NULL) {   //empty ?
        cout << "** WARNING - RetrieveQ() - No elements on list **";
     }

     nq = NQ();
     if( TRACE ) printf(" Number in Queue : %d\n", nq);
     NQAccum->SumDstat( (float) NQ() );  // Collect Nq statistics

     Qlist->Start();
     while ( (p = Qlist->Forward() ) != NULL ) {
        jobp = (Job *) p->info;
        taskp = jobp->CurrTask;
        if( TRACE ) printf(" Current searched task %d \n", taskp->num);
        if (taskp->num == ctnum)
        {
           reqelm = p;
           found = 1;
        }
     }

     if( TRACE ) printf(" Value of found : %d \n", found);

     if ( found == 1){
        jobp = (Job *) reqelm->info;
        taskp = jobp->CurrTask;
        if( TRACE ) printf(" Removed task number is : %d \n", taskp->num);
        Qlist->RmElem( reqelm );
     }
     else
     {
        jobp = NULL;
     }
 }
 break;

    default:                 // STATIC PRIORITY, FIFO, etc. default
      jobp = (Job *) p->info;
      Qlist->RmElem( p );    // remove the element (not info) from the list
    break;
  } // end switch

  return jobp;             // none found
} // end RetrieveQ()


//================================================================== NQ()
//  Retrieve the number in the queue

int Queue::NQ( void ) { return Qlist->Get_ItemCount();}

//============================================================ PrntNQStats()
//  Print current stats for the number in the queue

void Queue::PrntNQStats( void ) {
  NQAccum->PrntDstat( (float) NQ() );
}

int Queue::InQueue( int tasknum )
{
  elem *p;
  Job *jobp;
  Task *taskp;
```

```
  int ctnum;

  ctnum = tasknum;  // assigned task number stored

  int found = 0;

   Qlist->Start();
   while ( (p = Qlist->Forward() ) != NULL ) {
       jobp = (Job *) p->info;
       taskp = jobp->CurrTask;
       if( TRACE ) printf(" Current searched task %d \n", taskp->num);
       if (taskp->num == ctnum)
           found = 1;
   } // end of while

     if( TRACE ) printf(" Value of found : %d \n", found);

  return( found );
} // end of function


// SIMPL Resource interface
#ifndef _RSRC_H
#define _RSRC_H 1

#define SIZE 50

#include "queue.h"

class Rsrc {    // Resource Class
public:
  Rsrc( char *name, int number, short capacity, int nproduct, int discipline,
  int (*comp) (void*, void*) );  // constructor
  ~Rsrc();                              // Destructor
  void PrntRsrc();  // Output Resource stats when called
  void Emptytsklst();  // Empty the task list
  char Name[20];    // Name of the resource (for output)
  int Num;          // Resource number (unique)
  Job *CurrJob;     // Current job being worked on
  short NMax;       // maximum identical (parallel units) available
  short NBusy;      // Number of busy units ( < MaxUnits)
  long NServed;     // Number Served
  Queue *Q;         // Queue used by the resource (list of queues in future?)
  DLList **tasklst;  // tasks assigned to this resource
  float Setup[10][10];  // Setup time array
  int AvailUnits[10]; // Available unit array
  int TaskOrder[SIZE];   // stores the task number to output solution
  int TOindex;
  float TotalIdle;       // stores the total idle time on the work center
// add a statistical accumulator later for Utilization?
};

#endif

// ----------------   Rsrc.cpp -------------------------
// Resources are needed to perform tasks ...
//
#include <general.h>     // General constants, etc.
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <iostream.h>

#include "rsrc.h"
int i, j, k, l;

//Constructor
```

```
Rsrc::Rsrc( char *name, int number, short capacity, int nproduct,
            int discipline, int (*comp) (void*, void*) ) {
  strcpy(Name, name );
  Num = number;
  CurrJob = NULL;
  NMax = capacity;
  NBusy = 0;              // default to idle
  Q = new Queue( strcat(name,"_Q"), INFINITY, discipline, comp );
  NServed = 0;
  TOindex = 0;  // initialize task order index to zero
  TotalIdle = 0.0;
  tasklst = new DLList*[capacity];  // set array of tasklists

  for(j=1;j<=capacity;j++){
    tasklst[j]= new DLList;
  }

  for(i=0;i<nproduct;i++){      // set array of setup times
    for(j=0;j<nproduct;j++){
        Setup[i][j] = 0.0;
    }
  }

  for (k=0; k<=capacity; k++){
        AvailUnits[k]=k;
  }

  for (l=0; l<SIZE; l++)
          TaskOrder[l] = 0;  // initializes the task order array elements to zero
}

Rsrc::~Rsrc(){delete Q; delete tasklst;};               // Destructor


//================================================================== PrntRsrc()
//  Prints Summary Statistics for Resources

void Rsrc::PrntRsrc() {
    printf("\n  %s", Name );
    printf("\n    Number Served: %5d", NServed );
} // end PrntRsrc()

//================================================================== Emptytsklst()
void Rsrc::Emptytsklst()
{
  int i, j;

  elem *p, *pn;

  for (j=1; j<=NMax; j++)
  {
     tasklst[j]->Start();
     p = tasklst[j]->Forward();
     while(p != NULL)
     {
       pn = tasklst[j]->Forward();
       tasklst[j]->RmElem(p);
       p = pn;
     }
  } // end of for - variable j

}  // end of Emptytsklst()


// SIMPL Solution interface for Local search
// Written by : Deepu Philip; 07/01/2003
#ifndef _SOLN_H
```

```
#define _SOLN_H 1

class Soln {    // Solution Class
public:
  Soln( float Cmx, int rows, int cols );  // constructor
  ~Soln();                              // Destructor
  void PrntSoln();  // Output Solution details when called
  float CMax;
  int NRes;
  int NTsk;
  int SlnMtx[10][50];   // stores the solution
  int MachTask[10];     // stores the index of current tasks
};

#endif

// --------------------------   soln.cpp -------------------------
// Solutions are the feasible set of task orders on each resources
// Written by : Deepu Philip; 07/02/2003
#include <general.h>     // General constants, etc.
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "soln.h"

//Constructor
Soln::Soln( float Cmx, int rows, int cols ) {
  int i, j;
  CMax = Cmx;
  NRes = rows;
  NTsk = cols;

  for (i=0; i<NRes; i++)
  {
    for (j=0; j<NTsk; j++)
      SlnMtx[i][j] = 0;     // initialize the contents of matrix to zero
  }

  for (i=0; i<NRes; i++)
     MachTask[i] = 0;

} // end of constructor

Soln::~Soln(){};               // Destructor

//================================================================ PrntSoln()
//  Prints Details of the solution upon request

void Soln::PrntSoln() {
  int k, l;
    printf("\n Solution Details ");
    printf("\n Current Maximum Completion Time : %.3f ", CMax );
    printf("\n Current Task Order : \n");

    for (k=0; k<NRes; k++)
    {
      for (l=0; l<NTsk; l++)
          printf(" %d", SlnMtx[k][l]);
      printf ("\n");
    }
} // end PrntSoln()


/* ---------------------------------- rfjs.cpp -----------------------------
    Program:  rfjs.cpp
    Description: A SIMPL simulation of a reentrant flexible jobshop;
```

```
                            problem descriptions are read in and simulated.
                            A local search built around the simulation
                            An object oriented approach using SIMPL C++ library
    SIMPL C++ Library provided by : Dr. Edward Mooney (Jobshop.cpp)
    Modified by : Deepu Philip for Master's Thesis
*/
// added random initial solution generation capability; ver 3.2; DP; 06/18/2003
#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>    // C++ I/O
#include <iomanip.h>
#include <math.h>
#include <string.h>

#include <calendar.h>     // Include calendar object
#include <tally.h>        // Observation-based stat. accumulator class
#include "rsrc.h"         // Rsrc class interface
#include "queue.h"        // Queue class interface
#include <dllist.h>       // Double-linked list class
#include "job.h"          // Job interface
#include <dstat.h>        // Discrete-change stat. accumulator class

#include <general.h>      // General constants, etc.

#define NSRVR 2           // Number of resources (and queues) in the model
#define WEBMODE FALSE
#define TASKMODE FALSE

unsigned int seed = 897651;

float Tnow;               // Current simulated time - global to all functions

// Jobshop Problem Parameters
int njob, nres, ntask, nproduct;
float aveops, aveloops, setupfactor, setupdeviation;
int Qdiscipline;

Rsrc **machines;            // indexed list of (pointers to) machines
Task **tasks;               // indexed list of (pointers to) tasks
Job **jobs;                 // indexed list of (pointers to) jobs

// System Statistics collection vars
 float TimeInSys;                   // Used to calculate time in system
 Tally *TimeInSysAccum;              // Number in system accumulators

 Calendar C(1000, 50);        // Create a calendar called C

FILE *infile, *outfile, *ganttfile, *solnfile;  // Data input and output file pointers
char datapath[80]="probinst/";  // problem AND search input/output datapath
char outpath[40]="simout/";     // output file path
char gpath[40]="ganttdata/";    // gantt bar chart data file
char spath[40]="initsoln/";     // initial solution file output path
char probnam[30]="";            // problem data set name
char dfile[48]="";              // problem data file (probnam.prb)
char ofile[48]="";              // output file ( if any )
char gfile[48]="";              // gantt data output file
char sfile[48]="";        // stores the solution file

// Function Prototypes (this file)
void IoSetup (int argc, char *argv[]);      // Open files, etc.
void Init( void );                          // Initialize calendar, etc.
void Report( void );                          // Initialize calendar, etc.

void Arrival( Job *jobp );     // Arrival event processing
void BegServ( Job *jobp );     // Begin service - Called from arrival & EndSrv
void EndServ( Job *jobp );     // End service event
```

```
int CmpJobLVF_Atime( Job *job1 , Job *job2 ); // compare jobs for queue insertion
float Rnd();            // returns a U(0,1) random number
short RandomServer( void );  // Sample a server number (.4, .6)

//=====================================================================  main()
int main(int argc, char *argv[] )
{
    int i;
    Event EventBuf;               // A temporary event buffer
    Job *jobp;

    float Tfin = 10000;   // Simulation run length ,,, (set big to stop when all jobs done)
    char Etype;

// ** End of declarations - Initialize variables **

    IoSetup( argc, argv );        // Open infile, outfile and prmfile
    Init();                       // Initialize a jobshop problem
    fclose( infile );

// ** Main Simulation Loop **

    while ( Tnow < Tfin && !C.empty() ) {
        EventBuf = C.remove();          // Get next event

        Tnow = EventBuf.Time;           // Save Event attributes from buffer
        Etype = EventBuf.Etype;
        jobp = (Job *) EventBuf.Einfo;

        if ( TRACE ) printf( " TNOW, Event Type = %5.2f, %c\n", Tnow, Etype );

        switch ( Etype ) {
            case 'A':  {          // ** Process an arrival
                Arrival( jobp );
                break;
            }
            case 'E':  {          // ** Process an End of Service
                EndServ( jobp );
                break;
            }
            default: {            // ** didn't match event type - error
                printf( "** Error - Event type %c not found ** \n", Etype );
                exit(2);
            }
        } // end switch

    } // end While

 printf("Completed Simulation. Writing to file %s....\n",ofile);

    fprintf( outfile, "** PROBLEM DATA SET: %s **\n", probnam);

    fprintf( outfile, "\n** JOB SHOP SIMULATION RESULTS **\n\n" );
    fprintf( outfile, "\nEnding Simulation System State" );
    fprintf( outfile, "\n    Tnow: %8.2f", Tnow );
    fprintf( outfile, "\n    Number in each queue: " );
    for ( i=0;i<nres; i++) fprintf( outfile, "%5d ", machines[i]->Q->NQ() );
    fprintf( outfile, "\n" );

#if WEBMODE
    printf( "\n\nSystem Statistics" );
    TimeInSysAccum->PrntTally();

    printf( "\n\nServer & Queue Statistics" );
    for ( i=0;i<nres; i++) {
        machines[i]->PrntRsrc();
        machines[i]->Q->PrntNQStats( );
```

```
    }
#endif

    printf("Cmax of the schedule = %8.2f \n", Tnow);
    printf("\n\n**Done**\n");

    Report();                       // Print Report

    delete []machines;
    delete []tasks;
    delete []jobs;

    return(0);
}

//******************** Event Functions ********************************
//
//================================================================== Arrival()
//  Process an arrival to the system; coded by Deepu Philip
//  End of service or an arrival both calls this function

void Arrival( Job *jobp )
{
    Task *taskp;
    Rsrc *machp;

    taskp = jobp->CurrTask;
    taskp->succlst->Start();         // initialize precedence lists at the start
    taskp->predlst->Start();

    machp = machines[taskp->rsrc];

    if ( TRACE ) printf( " ARRIVAL      : %s, Job Number %d, Task Number %d\n",
                                         machp->Name, jobp->Num, taskp->num );

    if ( machp->NBusy >= machp->NMax )
        machp->Q->InsertQ( jobp );

    else
        BegServ( jobp );

    return;
} // End Arrival()

//
//================================================================== BegServ()
//  Begin service routine begins the service of the job;
//  depends what is in queue and which machine is free; coded by: deepu philip

void BegServ( Job *jobp )
{
    float TEvent;

    Task *taskp, *tasklp;
    Rsrc *machp;
    elem *p;
    Job *joblp;
    int prodnumi, prodnumj;
    float tsetupij=10000000.0;
    int tempunit;
    float tempsetupij = 0.0;
    int munit, capacity, i;
    int tindex;

    taskp = jobp->CurrTask;
    machp = machines[taskp->rsrc];
    //munit = machp->NBusy + 1;       // Assume seize last unit completed
```

```
    capacity = machp->NMax;
    for (i=capacity; i>=1;i--){       // Find lowest setup time to choose unit
        if (machp->AvailUnits[i]>0){
                tempunit = i;
                machp->tasklst[tempunit]->Start();
                if ( ( p = machp->tasklst[tempunit]->Backward() ) != NULL ) {

                                tasklp = (Task *) p->info;    //Get last task on machine list
                                joblp=jobs[tasklp->job];
                                prodnumj = jobp->Prodnum;
                                prodnumi = joblp->Prodnum;
                                tempsetupij=machp->Setup[prodnumi][prodnumj];
                }
                if (tempsetupij <= tsetupij){   // Check if smallest setup time
                                tsetupij = tempsetupij;
                                munit = tempunit;
                }
        }
    }

    machp->AvailUnits[munit]=-1;

    tindex = machp->TOindex;                 // obtain index
    machp->TaskOrder[tindex] = taskp->num;   // store task number
    machp->TOindex++;                  // increment the task index

                // list not empty => not first task on machine

    if ( TRACE ) printf( " BEGIN SERVICE: %s, Job Number %d, Task Number %d\n",
                                          machp->Name, jobp->Num, taskp->num );

// Schedule the end of service
    taskp->TStart = Tnow;        // Set task start time

    TEvent = Tnow + taskp->dur + tsetupij;
    C.Schedule( 'E', TEvent, jobp );

    machp->CurrJob = jobp;
    machp->NBusy++;                // Set another server busy
    machp->tasklst[munit]->Append(taskp);

} // End BegServ()

//
//================================================================== EndServ()
//  Process an end of service event; where the delay is completed
//  end of a service frees the machines; coded by deepu philip

void EndServ( Job *jobp )
{
    Task *taskp, *task2;
    Rsrc *machp;

    elem *p;                          // list element pointer
    int capacity, i;

    taskp = jobp->CurrTask;
    machp = machines[taskp->rsrc];
    taskp->TEnd = Tnow;       // Set task end time

    capacity = machp->NMax;
    for (i=1;i<=capacity;i++){
        machp->tasklst[i]->Start();
        while ( ( p = machp->tasklst[i]->Forward() ) != NULL ) {
                task2 = (Task *) p->info;
                if (task2->num == taskp->num){
```

```
                        machp->AvailUnits[i] = i;
                }
        }
    }

    if ( TRACE ) printf( " END SERVICE  : %s, Job Number %d, Task Number %d\n",
                                            machp->Name, jobp->Num, taskp->num );

    if  ( ( p = taskp->succlst->Forward() ) != NULL ) {
        taskp = (Task *) p->info;                   // schedule next task to arrive
        if ( taskp->num == ntask - 1 ) {
                                // successor dummy end task => last task for the job?
            TimeInSys = Tnow - machp->CurrJob->Atime;    // Time in system
            TimeInSysAccum->SumTally( TimeInSys );       // Update Accumulator
        jobp->JEndTime = Tnow;                 // Save job end time
        }
        else {
                jobp->CurrTask = taskp;                     // Schedule next task to arrive
                C.Schedule( 'A', Tnow, jobp );
        }
    }
    else {
        printf( "**ERROR: EndServ() - No Successor **" );
                                // only 1 EndofService per task
    }

    machp->NServed++;                           // No. served counter

    machp->NBusy--;           // Reduce number of busy servers

    if ( machp->Q->NQ() > 0 ) {                 // Any jobs to serve on machp?
        jobp = machp->Q->RetrieveQ();
        BegServ( jobp );
    }

} // End EndServ()


//********************* Other Functions ********************************

//========================================================= Cmp_JobLVF_Atime()
//Function compares two jobs based on the arrival time for
//      FIFO (LVF on arrival time)
//  return -1 if job1->Atime > job2->Atime
//  return  1 if job1->Atime < job2->Atime
//  return  0 if job1->Atime = job2->Atime
//
int CmpJobLVF_Atime( Job *job1 , Job *job2 )
{
        if ( job1->Atime > job2->Atime )
                return -1;
        else if ( job1->Atime < job2->Atime )
                return 1;
        else
                return 0;

}//end CmpJobLVF_Atime()

//=========================================================================== Rnd()
//  Returns a Uniform random number between 0 and 1

float Rnd( void ) {

    return (float) rand() / (float) RAND_MAX;

} // end Rnd()
```

```
//=================================================================RandomServer()
//  Returns a random server with p(0)=.4, p(1) = .6

short RandomServer( void ) {
    short Serveri;

    Serveri = 0;                    // Get server - arrays dimensioned from 0
    if ( Rnd() > .4 ) Serveri = 1;
    return Serveri;
} // end RandomServer()


// ------------------------------------------------------------- IoSetup ()
// sets up the initialization and file handling process
// coded by deepu philip
void IoSetup (int argc, char *argv[])
{
     int value;    // switch variable

// open input file
    if ( argc > 1 ) {
         if ( !strcmp( argv[1], "-?" ) ) {
                printf( "\n***** USAGE: rffss problem_file_name queue_discipline \n\n" );
                exit(0);
         }
         else
             {
                strcpy( probnam, argv[1] );   // obtain the problem name
                    value = atoi(argv[2]);  // converts into integer value - stdlib function
                    switch (value)    // decide which queue discipline
                    {
                        case 1: Qdiscipline = value;
                                    break;
                        case 2: Qdiscipline = value;
                                    break;
                        case 3: Qdiscipline = value;
                                    break;
                        case 4: Qdiscipline = value;
                                    break;
                        case 5: Qdiscipline = value;
                                    break;
                        case 6: Qdiscipline = value;
                                    break;
                     default: printf("Input the Queue Discipline : ");
                                    scanf("%d", &Qdiscipline);
                                    break;
                    }  // end of switch statement
             }  // end of else
    }
    else {
         printf("Input -> jobshop problem dataset name (.prb extension assumed): ");
         gets( probnam );            // .prb extension assumed ... don't input!
             printf("Input the Queue Discipline : ");
             scanf("%d", &Qdiscipline);
    }

    sprintf( dfile, "%s%s.prb", datapath, probnam );
    if((infile=fopen(dfile,"r"))==NULL)
            {printf("%s open failed\n",dfile);
                exit(-1);
            }
    else printf("%s opened. \n",dfile);

    sprintf( ofile, "%s%s.out", outpath, probnam );
    if((outfile=fopen(ofile,"w"))==NULL) {
        printf("%s open failed\n",ofile);
```

```
        exit(-1);
    };

    sprintf( gfile, "%s%s.txt", gpath, probnam );
    if((ganttfile=fopen(gfile,"w"))==NULL) {
        printf("%s open failed\n",gfile);
        exit(-1);
    };

    sprintf( sfile, "%s%s.sln", spath, probnam );
    if((solnfile = fopen(sfile, "w")) == NULL)
    {
        printf("%s open failed \n", sfile);
        exit(-1);
    };

    fflush(stdout);

} /* end IoSetup() */

// ------------------------------------------------------------------ Init()
// Description:
//   Init() reads input data and initializes the event calendar
//   modified by deepu philip;

void Init( void )
{
    Task *taskp, *adjtask, *first, *last, *prev, *next;
    Rsrc *machp;
    Job *jobp;

    elem *p;               // a DLList element

    int buflen = 130;
    int i,j,k,l,it;
    float setuptime;
    int nprocesses,ntask1;
    int nunit, prodnum;
    float ArrivalTime, DueDate;
    // int Qdiscipline;
    float MTTS, totalsetup, avesetup;
    int numsteps;

    char probname[101];
    int machine, ptime;
    char ChBuf[100];              // Temporary buffer for names & titles

    Tnow = 0.0;                     // Simulation starts at time zero
    srand(seed);                    // Initialize the random number stream


    // Create & initialize system statistics accumulators

    sprintf( ChBuf, "Time in the System" );    // Flow time accumulators
    TimeInSysAccum = new Tally( ChBuf );

// ** Input the Problem Data and Initialize Resources, Jobs and Tasks **

    fgets( probname, 100, infile );
    fscanf( infile, "%d %d %d", &njob, &nproduct, &nres);
    printf(" %d jobs, %d products, %d resources. \n", njob, nproduct, nres);

    jobs = new Job * [njob];         // create array of pointers to jobs

    machines = new Rsrc *[nres];    // create pointers to machines
    for (j = 0; j < nres; j++) {   // create machines
        sprintf( ChBuf, "Machine %d", j );
```

```
        fscanf(infile, "%d", &nunit);
        machines[j] = new Rsrc( ChBuf, j, nunit, nproduct, Qdiscipline,
                                            (int (*) (void *, void *)) CmpJobLVF_Atime );

        for(i=0;i<nproduct;i++){
                for(k=0;k<nproduct;k++){
                                fscanf(infile, "%f", &setuptime);
                        machines[j]->Setup[i][k]=setuptime;
                }
        }
    }

    fprintf(solnfile, "%d\n", nres);  // print the no of machines

    fscanf( infile, "%d", &ntask1);   // get the total number of tasks
    ntask = ntask1 + 2;
    tasks = new Task *[ntask];       // create array of pointers to tasks

 /* == Input  Info == */
#if WEBMODE
    printf ("\nInitializing ==> %d Job Steps\n", ntask);
#endif

    it = 0;  /* task index */
    first = tasks[0] = new Task( 0 );   // first and last (dummy) task pointers
    last = tasks[ntask - 1] = new Task( ntask-1 );

    for (i = 0; i < njob; i++) {    // read and initialize actual tasks
        jobs[i] = new Job(i);         // create job object
        jobp = jobs[i];

        fscanf(infile, "%d", &prodnum);     // read the product number
        jobp->Prodnum = prodnum;

        fscanf(infile, "%d", &nprocesses);  // read the number of processes
        jobp->jobntasks = nprocesses;

        for (j = 0; j<jobp->jobntasks; j++) {  // job steps - tasks for each job

            it++;
            fscanf (infile, "%d %d", &machine, &ptime);

            taskp = new Task( it );         // create the task object
            if( taskp == NULL) {
            printf(
                        "** ERROR - Init() - Insufficient memory for task[%d] **", it );
                    exit( 1 );
            }

            taskp->num = it;
            taskp->dur = (float) ptime;
            taskp->rsrc = machine;
            taskp->job = i;
            taskp->jobstep = j;

            tasks[it] = taskp;

            machp = machines[machine];
            //machp->tasklst->Append( taskp );  /* place on the machine list later*/

        }  /* next j (machine) */

    } /* next i (job)  */

//** Assign an average setup time for each job on each resource **
    if(nproduct>1){
        k = 0;
```

```
        for (i=0;i<njob;i++){
        jobp = jobs[i];
        k = jobp->Prodnum;
        for (j=0;j<nres;j++){
                machp = machines[j];
                for (l=0;l<nproduct;l++){
                        totalsetup = totalsetup + machp->Setup[k][l];
                }
                avesetup = totalsetup/(nproduct - 1);   // don't include same product
                jobp->AveSetup[j] = avesetup;
                totalsetup = 0.0;
        }
    }
    }
    else{
        for (i=0;i<njob;i++){
        jobp = jobs[i];
        jobp->AveSetup[j] = 0.0;
            }
    }

// ** Assign a MTTS to each task **

    for (i=njob-1;i>=0;i--){
        MTTS = 0.0;
        numsteps = 0;
        jobp = jobs[i];
        for (j=ntask - 2;j>=0;j--){
                taskp = tasks[j];
                if (jobp->Num == taskp->job){
                        MTTS = MTTS + taskp->dur;
                            numsteps = numsteps + 1;
                        taskp->MTTS = MTTS;
                        taskp->numsteps = numsteps;
                            taskp->RemTime = MTTS;
                }
        }
    }


// ** Create the precedence network task links **
    it = 0;
    for (i = 0; i < njob; i++) {
        jobp = jobs[i];
        for (j = 0; j < jobp->jobntasks; j++) {
            it++;
            taskp = tasks[it];

            if ( j == 0) {          /* first jobstep - link to dummy start node */
                jobp->CurrTask = taskp;        // initialize the jobstep for this job
            jobp->FirstTask = taskp;
                prev = first;
                next = tasks[it+1];
                first->succlst->Append(taskp);
            }
            else if ( j == jobp->jobntasks - 1 ) {  /* last jobstep? */
                prev = tasks[it-1];
                next = last;
                last->predlst->Append(taskp);
            }
            else {
                prev = tasks[it-1];
                next = tasks[it+1];
            }

            taskp->predlst->Append(prev);
            taskp->succlst->Append(next);
```

```
        }  /* next j (machine) */

    } /* next i  (job) */


#if DEBUG
 // for ( j=0; j<nres; j++ ) {         // output machine-task assignment lists */
 //   machp = machines[j];
 //   fprintf( outfile, "Machine %d Tasks : ", machp->Num );
 //   machp->tasklst->Start();

 //   while ( (p = machp->tasklst->Forward() ) != NULL ) {
 //     taskp = (Task *) p->info;
 //     fprintf( outfile, " %d %d %d : ", taskp->num, taskp->job, taskp->jobstep );
 //   }
 //   fprintf( outfile, "\n" );
 // }

    for ( it=0; it < ntask; it++ ) {   // Output precedences by task
        taskp=tasks[it];
        fprintf( outfile, "Task %d : %f, %d, %d, %d\n",
                        taskp->num, taskp->dur, taskp->rsrc, taskp->job, taskp->jobstep );

        fprintf( outfile, "   Successors: " );
        taskp->succlst->Start();
        while ( ( p = taskp->succlst->Forward() ) != NULL ) {
            adjtask = (Task *) p->info;
            fprintf( outfile, " %d %d %d : ", adjtask->num, adjtask->job, adjtask->jobstep );
        }

        fprintf( outfile, "\n   Predecessors: " );
        taskp->predlst->Start();
        while ( (p = taskp->predlst->Forward() ) != NULL ) {
            adjtask = (Task *) p->info;
            fprintf( outfile, " %d %d %d : ", adjtask->num, adjtask->job, adjtask->jobstep );
        }
        fprintf( outfile, "\n" );
    }

#endif

//** Schedule initial arrivals for each job **
    for (i = 0; i < njob; i++) {
        jobp = jobs[i];
        taskp = jobp->CurrTask;
        fscanf(infile, "%f", &ArrivalTime);
        C.Schedule( 'A', ArrivalTime, (void *) jobp );    // Schedule 1st Arrival

        jobp->Atime = ArrivalTime;         // arrival (release) time

     }

//** Schedule due dates for each job **
    for (i = 0; i < njob; i++) {
        jobp = jobs[i];
        taskp = jobp->CurrTask;
        fscanf(infile, "%f", &DueDate);
        jobp->DueDate = DueDate;         // arrival (release) time

     }

//** Read in the number of average ops/loop and loops **

     fscanf(infile, "%f %f", &aveops, &aveloops);
     fscanf(infile, "%f %f", &setupfactor, &setupdeviation);
```

```
printf("Completed the Init Function. Doing the Simulation Now...... \n");

} /* end Init() */


// ------------------------------------------------------------------ Report()
// Description:
//   Report() prints output to file; which is the schedule based on
//   dispatch rule.

void Report( void )
{
    Task *taskp, *task2, *adjtaskp, *first, *last, *prev, *next;
    Rsrc *machp;
    Job *jobp;
    int tskcnt;

    elem *p;               // a DLList element
    int i,j,k,l;
    float Diff, Diff1;
    int capacity;
    float late=0.0;
    float TotalLateness = 0.0;
    float AveLateness, AveFlowTime, FlowTime;
    float TotalFlowTime = 0.0;
    float MaxLate = 0.0;
    float PercentLate, MaxFlowTime;

    fprintf( outfile, "\n");
    fprintf( outfile, "Schedule by Job:\n");
    fprintf( outfile, "\n");

    for ( i=0; i < njob; i++ ) {   // Output schedule by job
        jobp = jobs[i];
        fprintf( outfile, "Job %d:\n", jobp->Num);
        for (j=1; j < ntask - 1; j++){
                taskp = tasks[j];
                if (taskp->job == jobp->Num){
                        machp = machines[taskp->rsrc];
                        capacity = machp->NMax;
                        for (k=1;k<=capacity;k++){
                                machp->tasklst[k]->Start();
                                while ( ( p = machp->tasklst[k]->Forward() ) != NULL ) {
                                        task2 = (Task *) p->info;
                                        if (taskp->num == task2->num){
                     fprintf( outfile, "Task %2d,Job %2d, ProdNum %d, Jobstep %2d,Machine %d,
                              Unit %d, Start, %3.2f, End, %3.2f \n", taskp->num, taskp->job,
                     jobp->Prodnum, taskp->jobstep, taskp->rsrc, k, taskp->TStart, taskp->TEnd );
#if !TASKMODE
                        fprintf( ganttfile, "%d,%d,%3.2f,%3.2f,", taskp->num, taskp->rsrc,
                        taskp->TStart, taskp->TEnd );
#endif
                                                                    }  // end of if
                                }  // end of while
                        }  // end of for - k
#if !TASKMODE
                if(j==(ntask - 1))
                     fprintf( ganttfile, "\n");
#endif
                } // end of for - j
        } // end of if
#if !TASKMODE
             if(i!=ntask)
                  fprintf( ganttfile, "\n");
#endif
    }  // end of for - i
```

```
    fprintf( outfile, "\n");
    fprintf( outfile, "Schedule by Machine:\n");
    fprintf( outfile, "\n");
    fprintf( solnfile, "%.2f\n", Tnow);

    for ( i=0; i < nres; i++ ) {   // Output schedule by resource
        machp = machines[i];
        fprintf( outfile, "Machine %d:\n", machp->Num);

        tskcnt = machp->TOindex;
        fprintf( solnfile, "%d %d\n", machp->Num, tskcnt);
                                                  // write machine number and task count
        for (l=0; l<tskcnt; l++)
                fprintf( solnfile, "%d ", machp->TaskOrder[l]);  // write to solution file
        fprintf( solnfile, "\n");

        for (j=1; j <= machp->NMax; j++){
        fprintf( outfile, "Unit %d:\n", j);
            machp->tasklst[j]->Start();

            while ( ( p = machp->tasklst[j]->Forward() ) != NULL ) {
                adjtaskp = (Task *) p->info;
                        fprintf( outfile, "Task  %2d,  Job  %2d,  Jobstep %2d,  Machine %d,
                        Unit %d,  Start  %3.2f,  End  %3.2f \n", adjtaskp->num, adjtaskp->job,
                        adjtaskp->jobstep,machp->Num,j, adjtaskp->TStart, adjtaskp->TEnd );
#if TASKMODE
                        fprintf( ganttfile, "%d,%3.2f,%3.2f,", adjtaskp->num, adjtaskp->TStart,
                        adjtaskp->TEnd );
#endif
             }
#if TASKMODE
                if(j!=machp->NMax)
                     fprintf( ganttfile, "\n");
#endif
        }
#if TASKMODE
             if(i!=(nres-1))
                   fprintf( ganttfile, "\n");
#endif
     fprintf( outfile, "\n");
    }

    fprintf( outfile, "\n");
    fprintf( outfile, "Comparison of Jobs with Lateness:\n");
    fprintf( outfile, "\n");
    for ( i=0; i < njob; i++ ) {   // Output lateness by job
        jobp = jobs[i];
        FlowTime = jobp->JEndTime - jobp->Atime;
#if WEBMODE
        printf("Job %d, Product %d, Flowtime, %f\n", i,jobp->Prodnum, FlowTime);
#endif
        TotalFlowTime = TotalFlowTime + FlowTime;
        if (FlowTime > MaxFlowTime){
                MaxFlowTime = FlowTime;
        }
        Diff = jobp->DueDate - jobp->JEndTime;
        if (Diff >= 0){
                fprintf( outfile, "Job %d,  End Time =, %.2f  Due Date =, %.2f  Early,
                                %.2f \n",jobp->Num, jobp->JEndTime, jobp->DueDate, Diff);
        }
        else{
        Diff1 = Diff * -1;
        fprintf( outfile, "Job %d,  End Time =, %.2f  Due Date =, %.2f  Late,
                         %.2f \n",jobp->Num, jobp->JEndTime, jobp->DueDate, Diff1);
        TotalLateness = TotalLateness + Diff1;
            late = late + 1;
        if (Diff1 > MaxLate){
```

```
                MaxLate = Diff1;
            }
            }
        }

        AveLateness = TotalLateness/late;
        AveFlowTime = TotalFlowTime/njob;
        PercentLate = late/njob*100;

        fclose( solnfile );
        fprintf( outfile, "\n");
        fprintf( outfile, "Completion time (Cmax) = %.2f \n",Tnow);
        fprintf( outfile, "Summary Explanation: probname, njobs, nproducts,
        nresources, Ave Ops, Ave Loops, Tnow, Average Flowtime, late, Average Lateness,
        Q-Discipline\n");
        fprintf( outfile, "**SUMMARY** %s, %d, %d, %d, %.1f, %.1f, %.2f, %.2f, %.2f, %.3f,%.3f,
        %.3f, %.3f, %.3f, %.3f, %d", probnam,njob, nproduct, nres, aveops, aveloops,setupfactor,
        setupdeviation,Tnow, AveFlowTime, MaxFlowTime, PercentLate, AveLateness,MaxLate,
        machp->Q->Discipline);

} // End Report

//================ example outputs ====================


Schedule by Job:

Job 0:
Task  1,Job  0, ProdNum 2, Jobstep  0, Machine 2, Unit 1, Start, 0.00, End, 60.00
Task  2,Job  0, ProdNum 2, Jobstep  1, Machine 0, Unit 1, Start, 60.00, End, 123.00
Task  3,Job  0, ProdNum 2, Jobstep  2, Machine 1, Unit 2, Start, 123.00, End, 174.00
Task  4,Job  0, ProdNum 2, Jobstep  3, Machine 2, Unit 2, Start, 182.00, End, 241.00
Task  5,Job  0, ProdNum 2, Jobstep  4, Machine 0, Unit 1, Start, 243.00, End, 304.00
Task  6,Job  0, ProdNum 2, Jobstep  5, Machine 1, Unit 1, Start, 329.00, End, 375.00
Job 1:
Task  7,Job  1, ProdNum 2, Jobstep  0, Machine 2, Unit 2, Start, 1.00, End, 61.00
Task  8,Job  1, ProdNum 2, Jobstep  1, Machine 0, Unit 2, Start, 61.00, End, 124.00
Task  9,Job  1, ProdNum 2, Jobstep  2, Machine 1, Unit 2, Start, 174.00, End, 225.00
Task 10,Job  1, ProdNum 2, Jobstep  3, Machine 2, Unit 2, Start, 363.00, End, 422.00
Task 11,Job  1, ProdNum 2, Jobstep  4, Machine 0, Unit 1, Start, 428.00, End, 489.00
Task 12,Job  1, ProdNum 2, Jobstep  5, Machine 1, Unit 1, Start, 506.00, End, 552.00
Job 2:
Task 13,Job  2, ProdNum 3, Jobstep  0, Machine 2, Unit 1, Start, 181.00, End, 242.00
Task 14,Job  2, ProdNum 3, Jobstep  1, Machine 1, Unit 1, Start, 260.00, End, 329.00
Task 15,Job  2, ProdNum 3, Jobstep  2, Machine 0, Unit 2, Start, 329.00, End, 381.00
Task 16,Job  2, ProdNum 3, Jobstep  3, Machine 2, Unit 2, Start, 422.00, End, 488.00
Task 17,Job  2, ProdNum 3, Jobstep  4, Machine 1, Unit 2, Start, 493.00, End, 555.00
Task 18,Job  2, ProdNum 3, Jobstep  5, Machine 0, Unit 2, Start, 574.00, End, 642.00
Job 3:
Task 19,Job  3, ProdNum 3, Jobstep  0, Machine 2, Unit 2, Start, 61.00, End, 122.00
Task 20,Job  3, ProdNum 3, Jobstep  1, Machine 1, Unit 1, Start, 122.00, End, 191.00
Task 21,Job  3, ProdNum 3, Jobstep  2, Machine 0, Unit 1, Start, 191.00, End, 243.00
Task 22,Job  3, ProdNum 3, Jobstep  3, Machine 2, Unit 1, Start, 361.00, End, 427.00
Task 23,Job  3, ProdNum 3, Jobstep  4, Machine 1, Unit 1, Start, 444.00, End, 506.00
Task 24,Job  3, ProdNum 3, Jobstep  5, Machine 0, Unit 2, Start, 506.00, End, 574.00
Job 4:
Task 25,Job  4, ProdNum 2, Jobstep  0, Machine 2, Unit 1, Start, 60.00, End, 120.00
Task 26,Job  4, ProdNum 2, Jobstep  1, Machine 0, Unit 1, Start, 123.00, End, 186.00
Task 27,Job  4, ProdNum 2, Jobstep  2, Machine 1, Unit 2, Start, 225.00, End, 276.00
Task 28,Job  4, ProdNum 2, Jobstep  3, Machine 2, Unit 1, Start, 302.00, End, 361.00
Task 29,Job  4, ProdNum 2, Jobstep  4, Machine 0, Unit 1, Start, 367.00, End, 428.00
Task 30,Job  4, ProdNum 2, Jobstep  5, Machine 1, Unit 2, Start, 447.00, End, 493.00
Job 5:
Task 31,Job  5, ProdNum 3, Jobstep  0, Machine 2, Unit 2, Start, 241.00, End, 302.00
Task 32,Job  5, ProdNum 3, Jobstep  1, Machine 1, Unit 2, Start, 327.00, End, 396.00
Task 33,Job  5, ProdNum 3, Jobstep  2, Machine 0, Unit 2, Start, 396.00, End, 448.00
Task 34,Job  5, ProdNum 3, Jobstep  3, Machine 2, Unit 2, Start, 554.00, End, 620.00
```

```
Task 35,Job  5, ProdNum 3, Jobstep  4, Machine 1, Unit 2, Start, 620.00, End, 682.00
Task 36,Job  5, ProdNum 3, Jobstep  5, Machine 0, Unit 2, Start, 682.00, End, 750.00
Job 6:
Task 37,Job  6, ProdNum 2, Jobstep  0, Machine 2, Unit 1, Start, 242.00, End, 302.00
Task 38,Job  6, ProdNum 2, Jobstep  1, Machine 0, Unit 1, Start, 304.00, End, 367.00
Task 39,Job  6, ProdNum 2, Jobstep  2, Machine 1, Unit 2, Start, 396.00, End, 447.00
Task 40,Job  6, ProdNum 2, Jobstep  3, Machine 2, Unit 1, Start, 486.00, End, 545.00
Task 41,Job  6, ProdNum 2, Jobstep  4, Machine 0, Unit 1, Start, 550.00, End, 611.00
Task 42,Job  6, ProdNum 2, Jobstep  5, Machine 1, Unit 1, Start, 611.00, End, 657.00
Job 7:
Task 43,Job  7, ProdNum 3, Jobstep  0, Machine 2, Unit 1, Start, 120.00, End, 181.00
Task 44,Job  7, ProdNum 3, Jobstep  1, Machine 1, Unit 1, Start, 191.00, End, 260.00
Task 45,Job  7, ProdNum 3, Jobstep  2, Machine 0, Unit 2, Start, 260.00, End, 312.00
Task 46,Job  7, ProdNum 3, Jobstep  3, Machine 2, Unit 2, Start, 488.00, End, 554.00
Task 47,Job  7, ProdNum 3, Jobstep  4, Machine 1, Unit 2, Start, 555.00, End, 617.00
Task 48,Job  7, ProdNum 3, Jobstep  5, Machine 0, Unit 1, Start, 617.00, End, 685.00
Job 8:
Task 49,Job  8, ProdNum 3, Jobstep  0, Machine 2, Unit 2, Start, 302.00, End, 363.00
Task 50,Job  8, ProdNum 3, Jobstep  1, Machine 1, Unit 1, Start, 375.00, End, 444.00
Task 51,Job  8, ProdNum 3, Jobstep  2, Machine 0, Unit 2, Start, 448.00, End, 500.00
Task 52,Job  8, ProdNum 3, Jobstep  3, Machine 2, Unit 1, Start, 545.00, End, 611.00
Task 53,Job  8, ProdNum 3, Jobstep  4, Machine 1, Unit 1, Start, 657.00, End, 719.00
Task 54,Job  8, ProdNum 3, Jobstep  5, Machine 0, Unit 1, Start, 719.00, End, 787.00
Job 9:
Task 55,Job  9, ProdNum 2, Jobstep  0, Machine 2, Unit 2, Start, 122.00, End, 182.00
Task 56,Job  9, ProdNum 2, Jobstep  1, Machine 0, Unit 2, Start, 182.00, End, 245.00
Task 57,Job  9, ProdNum 2, Jobstep  2, Machine 1, Unit 2, Start, 276.00, End, 327.00
Task 58,Job  9, ProdNum 2, Jobstep  3, Machine 2, Unit 1, Start, 427.00, End, 486.00
Task 59,Job  9, ProdNum 2, Jobstep  4, Machine 0, Unit 1, Start, 489.00, End, 550.00
Task 60,Job  9, ProdNum 2, Jobstep  5, Machine 1, Unit 1, Start, 552.00, End, 598.00

Schedule by Machine:

Machine 0:
Unit 1:
Task   2,  Job   0,  Jobstep  1,  Machine 0,  Unit 1,  Start  60.00,  End  123.00
Task  26,  Job   4,  Jobstep  1,  Machine 0,  Unit 1,  Start  123.00,  End  186.00
Task  21,  Job   3,  Jobstep  2,  Machine 0,  Unit 1,  Start  191.00,  End  243.00
Task   5,  Job   0,  Jobstep  4,  Machine 0,  Unit 1,  Start  243.00,  End  304.00
Task  38,  Job   6,  Jobstep  1,  Machine 0,  Unit 1,  Start  304.00,  End  367.00
Task  29,  Job   4,  Jobstep  4,  Machine 0,  Unit 1,  Start  367.00,  End  428.00
Task  11,  Job   1,  Jobstep  4,  Machine 0,  Unit 1,  Start  428.00,  End  489.00
Task  59,  Job   9,  Jobstep  4,  Machine 0,  Unit 1,  Start  489.00,  End  550.00
Task  41,  Job   6,  Jobstep  4,  Machine 0,  Unit 1,  Start  550.00,  End  611.00
Task  48,  Job   7,  Jobstep  5,  Machine 0,  Unit 1,  Start  617.00,  End  685.00
Task  54,  Job   8,  Jobstep  5,  Machine 0,  Unit 1,  Start  719.00,  End  787.00
Unit 2:
Task   8,  Job   1,  Jobstep  1,  Machine 0,  Unit 2,  Start  61.00,  End  124.00
Task  56,  Job   9,  Jobstep  1,  Machine 0,  Unit 2,  Start  182.00,  End  245.00
Task  45,  Job   7,  Jobstep  2,  Machine 0,  Unit 2,  Start  260.00,  End  312.00
Task  15,  Job   2,  Jobstep  2,  Machine 0,  Unit 2,  Start  329.00,  End  381.00
Task  33,  Job   5,  Jobstep  2,  Machine 0,  Unit 2,  Start  396.00,  End  448.00
Task  51,  Job   8,  Jobstep  2,  Machine 0,  Unit 2,  Start  448.00,  End  500.00
Task  24,  Job   3,  Jobstep  5,  Machine 0,  Unit 2,  Start  506.00,  End  574.00
Task  18,  Job   2,  Jobstep  5,  Machine 0,  Unit 2,  Start  574.00,  End  642.00
Task  36,  Job   5,  Jobstep  5,  Machine 0,  Unit 2,  Start  682.00,  End  750.00

Machine 1:
Unit 1:
Task  20,  Job   3,  Jobstep  1,  Machine 1,  Unit 1,  Start  122.00,  End  191.00
Task  44,  Job   7,  Jobstep  1,  Machine 1,  Unit 1,  Start  191.00,  End  260.00
Task  14,  Job   2,  Jobstep  1,  Machine 1,  Unit 1,  Start  260.00,  End  329.00
Task   6,  Job   0,  Jobstep  5,  Machine 1,  Unit 1,  Start  329.00,  End  375.00
Task  50,  Job   8,  Jobstep  1,  Machine 1,  Unit 1,  Start  375.00,  End  444.00
Task  23,  Job   3,  Jobstep  4,  Machine 1,  Unit 1,  Start  444.00,  End  506.00
Task  12,  Job   1,  Jobstep  5,  Machine 1,  Unit 1,  Start  506.00,  End  552.00
Task  60,  Job   9,  Jobstep  5,  Machine 1,  Unit 1,  Start  552.00,  End  598.00
```

```
Task  42,  Job   6,  Jobstep  5,  Machine 1,  Unit 1,  Start  611.00,  End  657.00
Task  53,  Job   8,  Jobstep  4,  Machine 1,  Unit 1,  Start  657.00,  End  719.00
Unit 2:
Task   3,  Job   0,  Jobstep  2,  Machine 1,  Unit 2,  Start  123.00,  End  174.00
Task   9,  Job   1,  Jobstep  2,  Machine 1,  Unit 2,  Start  174.00,  End  225.00
Task  27,  Job   4,  Jobstep  2,  Machine 1,  Unit 2,  Start  225.00,  End  276.00
Task  57,  Job   9,  Jobstep  2,  Machine 1,  Unit 2,  Start  276.00,  End  327.00
Task  32,  Job   5,  Jobstep  1,  Machine 1,  Unit 2,  Start  327.00,  End  396.00
Task  39,  Job   6,  Jobstep  2,  Machine 1,  Unit 2,  Start  396.00,  End  447.00
Task  30,  Job   4,  Jobstep  5,  Machine 1,  Unit 2,  Start  447.00,  End  493.00
Task  17,  Job   2,  Jobstep  4,  Machine 1,  Unit 2,  Start  493.00,  End  555.00
Task  47,  Job   7,  Jobstep  4,  Machine 1,  Unit 2,  Start  555.00,  End  617.00
Task  35,  Job   5,  Jobstep  4,  Machine 1,  Unit 2,  Start  620.00,  End  682.00

Machine 2:
Unit 1:
Task   1,  Job   0,  Jobstep  0,  Machine 2,  Unit 1,  Start  0.00,  End  60.00
Task  25,  Job   4,  Jobstep  0,  Machine 2,  Unit 1,  Start  60.00,  End  120.00
Task  43,  Job   7,  Jobstep  0,  Machine 2,  Unit 1,  Start  120.00,  End  181.00
Task  13,  Job   2,  Jobstep  0,  Machine 2,  Unit 1,  Start  181.00,  End  242.00
Task  37,  Job   6,  Jobstep  0,  Machine 2,  Unit 1,  Start  242.00,  End  302.00
Task  28,  Job   4,  Jobstep  3,  Machine 2,  Unit 1,  Start  302.00,  End  361.00
Task  22,  Job   3,  Jobstep  3,  Machine 2,  Unit 1,  Start  361.00,  End  427.00
Task  58,  Job   9,  Jobstep  3,  Machine 2,  Unit 1,  Start  427.00,  End  486.00
Task  40,  Job   6,  Jobstep  3,  Machine 2,  Unit 1,  Start  486.00,  End  545.00
Task  52,  Job   8,  Jobstep  3,  Machine 2,  Unit 1,  Start  545.00,  End  611.00
Unit 2:
Task   7,  Job   1,  Jobstep  0,  Machine 2,  Unit 2,  Start  1.00,  End  61.00
Task  19,  Job   3,  Jobstep  0,  Machine 2,  Unit 2,  Start  61.00,  End  122.00
Task  55,  Job   9,  Jobstep  0,  Machine 2,  Unit 2,  Start  122.00,  End  182.00
Task   4,  Job   0,  Jobstep  3,  Machine 2,  Unit 2,  Start  182.00,  End  241.00
Task  31,  Job   5,  Jobstep  0,  Machine 2,  Unit 2,  Start  241.00,  End  302.00
Task  49,  Job   8,  Jobstep  0,  Machine 2,  Unit 2,  Start  302.00,  End  363.00
Task  10,  Job   1,  Jobstep  3,  Machine 2,  Unit 2,  Start  363.00,  End  422.00
Task  16,  Job   2,  Jobstep  3,  Machine 2,  Unit 2,  Start  422.00,  End  488.00
Task  46,  Job   7,  Jobstep  3,  Machine 2,  Unit 2,  Start  488.00,  End  554.00
Task  34,  Job   5,  Jobstep  3,  Machine 2,  Unit 2,  Start  554.00,  End  620.00


Comparison of Jobs with Lateness:

Job 0,  End Time =, 375.00  Due Date =, 405.00  Early, 30.00
Job 1,  End Time =, 552.00  Due Date =, 406.00  Late, 146.00
Job 2,  End Time =, 642.00  Due Date =, 461.00  Late, 181.00
Job 3,  End Time =, 574.00  Due Date =, 462.00  Late, 112.00
Job 4,  End Time =, 493.00  Due Date =, 409.00  Late, 84.00
Job 5,  End Time =, 750.00  Due Date =, 464.00  Late, 286.00
Job 6,  End Time =, 657.00  Due Date =, 411.00  Late, 246.00
Job 7,  End Time =, 685.00  Due Date =, 466.00  Late, 219.00
Job 8,  End Time =, 787.00  Due Date =, 467.00  Late, 320.00
Job 9,  End Time =, 598.00  Due Date =, 414.00  Late, 184.00

Completion time (Cmax) = 787.00
Summary Explanation: probname, njobs, nproducts, nresources, Ave Ops, Ave Loops,
Tnow, Average Flowtime, late, Average Lateness, Q-Discipline
**SUMMARY** testprobm, 10, 3, 3, 3.0, 2.0, 0.20, 0.10, 787.00, 606.800,779.000, 90.000,
197.556, 320.000, 7

//================= initial solution  =================

3
787.00
0 20
2 8 26 56 21 5 45 38 15 29 33 11 51 59 24 41 18 48 36 54
1 20
20 3 9 44 27 14 57 32 6 50 39 23 30 17 12 60 47 42 35 53
2 20
```

1 7 25 19 43 55 13 4 31 37 49 28 22 10 16 58 40 46 34 52

APPENDIX C - PERFORMANCE EVALUATION

Different programs were written for converting various restrictions of RFJSSDS to the format that to be read by the simulation. The details are given below.

```
//---------------------------------------- stdjshop.cpp -----------------------------
// Program Description:
//     Generates the prb files for the standard job shop problems for thesis
//     Works for all test problems of Applegate.
// coded by: Deepu Philip, 02/10/2003

#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <fstream.h>
#include <ctype.h>
#include <iomanip.h>
#include <stdio.h>
using namespace std;  // for handling strings

#define SIZE 50
#define DEBUG TRUE

struct Jobstr       // structure to hold the job details
{
    int num;          // store job number
    int arrival;      // arrival time of the job
    int duedate;      // due date of the job
    int seqarray[SIZE+1];  // job sequence or route
    int ptimearray[SIZE+1]; // proessing time array
};


char outpath[20]="jshopinst/";     // output file path
char ofile[40]="";                 // output file
char inpath[20]="appleprobs/";     // input file path
char ifile[40]="";                 // input file

FILE *fin, *fout;   // input and output file pointer

long seed = 12345;

int main(int argc, char *argv[])   // allows to pass the arguments by command line
{
    int njob, nmach, nprod, capacity;
    int i, j, k;
    int dloopsize;
    int tracker;
    int rcount, ptcount, tcount, nprocess;
    int cumptime;
    float ops, loops, setupsevfact, setupdeviation;

    if (argc < 1)  // missing input and output file name?
    {
        cout <<" Usage: stdjshop <input filename> (.prb extension assumed)" << endl;
        exit(1);
    } // end of miss check

    if (!strcmp(argv[1], "-?"))  // user want to know how to use?
    {
        cout <<" USAGE: stdjshop <input filename> (.prb extension assumed)" << endl;
        exit(1);
    }  // end of user know check
```

```
sprintf(ifile, "%s%s.prb", inpath, argv[1]);  // get the input file with path
if((fin=fopen(ifile,"r"))==NULL)  // can input file be opened?
{
    cout << "Input file " << ifile << " opening failed." << endl;
    exit(1);
}
else
    cout << ifile << " opened successfully for reading." << endl;

char prbtitle[100];  // title is an array of characters
fgets(prbtitle, 100, fin);  // get the problem title
cout << "Reading " << prbtitle << endl;
fscanf(fin, "%d %d", &njob, &nmach);   // get number of jobs and machines
nprod = njob;  // each job is a product
ops = (float) nmach; // operations equal number of machines
loops = 1.0;   // only one loop
capacity = 1;  // capacity is 1 always

// generate the outfile name as per needed for further parsing
sprintf(ofile, "%s%s_%d%d%1.0f%d_%ld.prb", outpath,argv[1],njob,nprod,loops,capacity,seed);

if((fout=fopen(ofile,"w"))==NULL)  // can output file be opened
{
    cout << "Output file " << ofile << " opening failed." << endl;
    exit(1);
}
else
    cout << ofile << " opened successfully for writing." << endl;

fprintf(fout, "%s\n", argv[1]);  // print name of problem to file

fprintf(fout, "%d %d %d\n", njob, nprod, nmach);  // jobs, products and machine info in file

int *resourcearray = new int[nmach+1];  // resource array - units of resource
for(i=1;i<=nmach;i++)
    resourcearray[i] = 1;    // how many units  - currently 1 for jobshop

for(i=1;i<=nmach;i++)   // for each resource
{
 fprintf(fout, "%d\n", resourcearray[i]);
 for(j=1;j<=nprod;j++)  // for each product
 {
     for(k=1;k<=nprod;k++)
         fprintf(fout, "%d ", 0);   // no sequence dependent setup times
     fprintf(fout, "\n");;
 }  // end of for loop (variable - j)
} // end of for loop (variable - i)

dloopsize = nmach*2;  // route information - (machine process-time) pair
tcount = 0;           // initialize task count
Jobstr *jobarray[njob+1];     // Array of structure for jobs
for(i=1;i<=njob;i++)
{
    jobarray[i] = new Jobstr;     // New instance of structure - dynamic allocation
    cumptime = 0;       // cumulative processing time
    jobarray[i]->arrival = 0;  // all jobs avialable at time 0
    jobarray[i]->num = i;
    tracker = 1;          // tracking variable for route and process time info
    rcount = 1;           // route index count
    ptcount = 1;          // processing time index cont
    for(j=1; j<=dloopsize; j++)
    {
        if((tracker % 2) != 0)  // if odd number
        {
             fscanf(fin, "%d", &jobarray[i]->seqarray[rcount]);
             rcount++;
             tcount++;
```

```
            }
            else                // even number
            {
                 fscanf(fin, "%d", &jobarray[i]->ptimearray[ptcount]);
                 cumptime = cumptime + jobarray[i]->ptimearray[ptcount];
                 ptcount++;
            }
            tracker++;
        }  // end of for loop - j
        jobarray[i]->duedate = cumptime;  // set the due date to cumulated processing time
        fscanf(fin, "\n");
 } // end of for loop - i

 cout << "Finished reading the data from " << ifile <<endl;
 fclose(fin);   // close input file

#if DEBUG
 cout << "no. of tasks :" << tcount << endl;
 for(i=1;i<=njob;i++)
 {
     cout << "Job :" << jobarray[i]->num << " Arrival :" << jobarray[i]->arrival << "
     Due Date :" << jobarray[i]->duedate << endl;
     for(j=1; j<=nmach; j++)
         cout << jobarray[i]->seqarray[j] << " " << jobarray[i]->ptimearray[j] << " ";
     cout << endl;
 }
#endif

 fprintf(fout, "%d\n", tcount);  // print how many tasks in the output file

 for(i=1;i<=njob;i++)   // machine # and process time for each job
 {
     fprintf(fout, "%d\n", jobarray[i]->num);   // write product number
     fprintf(fout, "%d\n", nmach);   // visits all machines (no. process = nmach)
     for(j=1;j<=nmach;j++)
         fprintf(fout, "%d %d ", jobarray[i]->seqarray[j], jobarray[i]->ptimearray[j]);
     fprintf(fout, "\n");
 } // end of for loop (variable - i)

 for(i=1;i<=njob;i++)   //arrival time for each job
        fprintf(fout, "%d\n", jobarray[i]->arrival);  // arrival time of each job in file

 for(i=1;i<=njob;i++)
        fprintf(fout, "%d\n", jobarray[i]->duedate); // due date of each job in file

 fprintf(fout, "%f %f\n", ops, loops);   // average operations and loops

 setupsevfact = 0.0;  // no set up severity factor
 setupdeviation = 0.0;  // no set up deviation factor
 fprintf(fout, "%f %f\n", setupsevfact, setupdeviation);

 cout << "Created " << ofile << " in simulation readable format." << endl;

 return 1;  // ansi c require main to return a value
} // end of main program

//========================== Result ==============================

abz9
20 20 15
1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
300
1
15
6 14 5 21 8 13 4 11 1 11 14 35 13 20 11 17 10 18 12 11 2 23 3 13 0 15 7 11 9 35
2
15
1 35 5 31 0 13 3 26 6 14 9 17 7 38 12 20 10 19 13 12 8 16 4 34 11 15 14 12 2 14
3
15
0 30 4 35 2 40 10 35 6 30 14 23 8 29 13 37 7 38 3 40 9 26 12 11 1 40 11 36 5 17
4
15
7 40 5 18 4 12 8 23 0 23 9 14 13 16 12 14 10 23 3 12 6 16 14 32 1 40 11 25 2 29
5
15
2 35 3 15 12 31 11 28 6 32 4 30 10 27 7 29 0 38 13 11 1 23 14 17 5 27 9 37 8 29
6
15
5 33 3 33 6 19 12 40 10 19 0 33 13 26 2 31 11 28 7 36 4 38 1 21 14 25 9 40 8 35
7
15
13 25 0 32 11 33 12 18 4 32 6 28 5 15 3 35 9 14 2 34 7 23 10 32 1 17 14 26 8 19
8
15
2 16 12 33 9 34 11 30 13 40 8 12 14 26 5 26 6 15 3 21 1 40 4 32 0 14 7 30 10 35
9
15
```

```
2 17 10 16 14 20 6 24 8 26 3 36 12 22 0 14 13 11 9 20 7 23 1 29 11 23 4 15 5 40
10
15
4 27 9 37 3 40 11 14 13 25 7 30 0 34 2 11 5 15 12 32 1 36 10 12 14 28 8 31 6 23
11
15
13 25 0 22 3 27 8 14 5 25 6 20 14 18 7 14 1 19 2 17 4 27 9 22 12 22 11 27 10 21
12
15
14 34 10 15 0 22 3 29 13 34 6 40 7 17 2 32 12 20 5 39 4 31 11 16 1 37 8 33 9 13
13
15
6 12 12 27 4 17 2 24 8 11 5 19 14 11 3 17 9 25 1 11 11 31 13 33 7 31 10 12 0 22
14
15
5 22 14 15 0 16 8 32 7 20 4 22 9 11 13 19 1 30 12 33 6 29 11 18 3 34 10 32 2 18
15
15
5 27 3 26 10 28 6 37 4 18 12 12 11 11 13 26 7 27 9 40 14 19 1 24 2 18 0 12 8 34
16
15
8 15 5 28 9 25 6 32 1 13 7 38 11 11 2 34 4 25 0 20 10 32 3 23 12 14 14 16 13 20
17
15
1 15 4 13 8 37 3 14 10 22 5 24 12 26 7 22 9 34 14 22 11 19 13 32 0 29 2 13 6 35
18
15
7 36 5 33 13 28 9 20 10 30 4 33 14 29 0 34 3 22 11 12 6 30 8 12 1 35 2 13 12 35
19
15
14 26 11 31 5 35 2 38 13 19 10 35 4 27 8 29 3 39 9 13 6 14 7 26 0 17 1 22 12 15
20
15
1 36 7 34 11 33 8 17 14 38 6 39 5 16 3 27 13 29 2 16 0 16 4 19 9 40 12 35 10 39
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
268
316
467
337
409
457
383
404
336
395
320
412
303
```

```
351
359
346
357
402
386
434
15.000000 1.000000
0.000000 0.000000


//--------------------------------------- rflowshop.cpp -----------------------------
// Program Description:
//     Generates the prb files for the reentrant flow shop problems for thesis
//     Works for all test problems of Demirkol and Uzsoy.

#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <fstream.h>
#include <ctype.h>
#include <iomanip.h>
#include <stdio.h>
using namespace std;

#define SIZE 50
#define DEBUG TRUE

struct Jobstr       // structure to hold the job details
{
  int num;          // store job number
  int arrival;      // arrival time of the job
  int duedate;      // due date of the job
  int seqarray[SIZE+1];  // job sequence or route
  int ptimearray[SIZE+1]; // proessing time array
};

char outpath[20]="rfsinst/";     // output file path
char ofile[40]="";                 // output file
char inpath[20]="uzoyprobs/";     // input file path
char ifile[40]="";                  // input file

FILE *fin, *fout;   // input and output file pointer

int main(int argc, char *argv[])   // allows to pass the arguments by command line
{
  int njob, nmach, nprod, nopers;
  int i, j, k;
  int dloopsize;
  int tracker;
  int rcount, ptcount, tcount, nprocess;
  int cumptime;
  float ops, loops, setupsevfact, setupdeviation;

  if (argc < 1)  // missing input and output file name?
  {
    cout <<" Usage: rflowshop <input filename> (.dat extension assumed)" << endl;
    exit(1);
  }

  if (!strcmp(argv[1], "-?"))  // user want to know how to use?
  {
    cout <<" USAGE: rflowshop <input filename> (.dat extension assumed)" << endl;
    exit(1);
  }

  sprintf(ifile, "%s%s.dat", inpath, argv[1]);
```

```
  if((fin=fopen(ifile,"r"))==NULL)  // can input file be opened?
  {
    cout << "Input file " << ifile << " opening failed." << endl;
    exit(1);
  }
  else
    cout << ifile << " opened successfully for reading." << endl;

  sprintf(ofile, "%s%s.prb", outpath, argv[1]);
  if((fout=fopen(ofile,"w"))==NULL)  // can output file be opened
  {
    cout << "Output file " << ofile << " opening failed." << endl;
    exit(1);
  }
  else
    cout << ofile << " opened successfully for writing." << endl;

  fprintf(fout, "%s\n", argv[1]);
  fscanf(fin, "%d %d %d", &nopers, &njob, &nmach);
  nprod = njob;  // each job is a product

  fprintf(fout, "%d %d %d\n", njob, nprod, nmach);

  int *resourcearray = new int[nmach+1];  // resource array - units of resource
  for(i=1;i<=nmach;i++)
    resourcearray[i] = 1;    // how many units  - currently 1 for flowshop

/*
  for(i=1;i<=nmach;i++)   // for each resource
  {
   fprintf(fout, "%d\n", resourcearray[i]);
   for(j=1;j<=nprod;j++)  // for each product
   {
     for(k=1;k<=nprod;k++)
       fprintf(fout, "%d ", 0);   // no sequence dependent setup times
     fprintf(fout, "\n");;
   }  // end of for loop (variable - j)
  } // end of for loop (variable - i)

  dloopsize = nmach*2;  // route information - (machine process-time) pair
  tcount = 0;           // initialize task count
  Jobstr *jobarray[njob+1];     // Array of structure for jobs
  for(i=1;i<=njob;i++)
  {
    jobarray[i] = new Jobstr;     // New instance of structure - dynamic allocation
    cumptime = 0;       // cumulative processing time
    jobarray[i]->arrival = 0;  // all jobs avialable at time 0
    jobarray[i]->num = i;
    tracker = 1;          // tracking variable for route and process time info
    rcount = 1;           // route index count
    ptcount = 1;          // processing time index cont
    for(j=1; j<=dloopsize; j++)
    {
      if((tracker % 2) != 0)  // if odd number
      {
         fscanf(fin, "%d", &jobarray[i]->seqarray[rcount]);
         rcount++;
         tcount++;
      }
      else                 // even number
      {
         fscanf(fin, "%d", &jobarray[i]->ptimearray[ptcount]);
         cumptime = cumptime + jobarray[i]->ptimearray[ptcount];
         ptcount++;
      }
      tracker++;
    }  // end of for loop - j
```

```
    jobarray[i]->duedate = cumptime;
    fscanf(fin, "\n");
 } // end of for loop - i

 cout << "Finished reading the data from " << ifile <<endl;

#if DEBUG
 cout << "no. of tasks :" << tcount << endl;
 for(i=1;i<=njob;i++)
 {
   cout << "Job :" << jobarray[i]->num << " Arrival :" << jobarray[i]->arrival <<
   " Due Date :" << jobarray[i]->duedate << endl;
   for(j=1; j<=nmach; j++)
     cout << jobarray[i]->seqarray[j] << " " << jobarray[i]->ptimearray[j] << " ";
   cout << endl;
 }
#endif

 fprintf(fout, "%d\n", tcount);

 for(i=1;i<=njob;i++)   // machine # and process time for each job
 {
   fprintf(fout, "%d\n", jobarray[i]->num);   // write product number
   fprintf(fout, "%d\n", nmach);   // visits all machines (no. process = nmach)
   for(j=1;j<=nmach;j++)
     fprintf(fout, "%d %d ", jobarray[i]->seqarray[j], jobarray[i]->ptimearray[j]);
   fprintf(fout, "\n");
 } // end of for loop (variable - i)

 for(i=1;i<=njob;i++)   //arrival time for each job
    fprintf(fout, "%d\n", jobarray[i]->arrival);

 for(i=1;i<=njob;i++)
    fprintf(fout, "%d\n", jobarray[i]->duedate);

 ops = (float) nmach;
 loops = 1.0;
 fprintf(fout, "%f %f\n", ops, loops);   // average operations and loops

 setupsevfact = 0.0;
 setupdeviation = 0.0;
 fprintf(fout, "%f %f\n", setupsevfact, setupdeviation);

*/
 cout << "Created " << ofile << " in simulation readable format." << endl;

 fclose(fin);   // close input file
 fclose(fout);  // close output file

 return 1;
}

//====================== Result ==========================

34 10 5
1 1 0 143 495 2
2 1 1 26 495 3
3 1 2 83 495 5
4 1 3 0 495 0
5 2 0 171 646 1
6 2 5 46 646 3
7 2 6 0 646 0
8 3 0 23 424 1
9 3 8 180 424 4
10 3 9 87 424 5
11 3 10 0 424 0
12 4 0 174 523 1
```

```
13 4 12 135 523 2
14 4 13 74 523 5
15 4 14 0 523 0
16 5 0 50 481 2
17 5 16 116 481 5
18 5 17 0 481 0
19 6 0 72 549 1
20 6 19 0 549 0
21 7 0 178 599 1
22 7 21 106 599 2
23 7 22 198 599 5
24 7 23 0 599 0
25 8 0 173 659 5
26 8 25 0 659 0
27 9 0 121 536 2
28 9 27 169 536 3
29 9 28 25 536 5
30 9 29 0 536 0
31 10 0 139 481 1
32 10 31 9 481 2
33 10 32 63 481 3
34 10 33 0 481 0
0 1 6
0 2 154
0 3 62
0 5 96
0 6 42
0 8 139
0 9 39
0 10 23
0 12 46
0 13 125
0 14 180
0 16 23
0 17 29
0 19 139
0 21 88
0 22 37
0 23 3
0 25 185
0 27 93
0 28 75
0 29 94
0 31 13
0 32 140
0 33 162
1 13 132
1 16 31
1 22 164
1 27 27
1 32 13
2 6 5
2 28 97
2 33 84
3 10 32
3 14 38
3 17 104
3 23 133
3 25 26
3 29 85
4 7 42
4 11 11
4 15 84
4 18 175
4 20 113
4 24 153
4 26 188
```

4 30 14
4 34 129
5 8 172
5 12 7
5 19 8
5 21 127
5 31 115
6 2 6
6 28 76
6 33 175
7 4 126
7 11 115
7 15 110
7 18 20
7 20 193
7 24 27
7 26 101
7 30 13
7 34 187
8 5 5
8 12 19
8 19 174
8 21 163
8 31 67
10 3 169
10 14 102
10 17 199
10 23 189
10 25 37
10 29 183
11 4 9
11 7 158
11 15 161
11 18 20
11 20 99
11 24 189
11 26 177
11 30 188
11 34 54
12 5 104
12 8 45
12 19 13
12 21 153
12 31 123
13 1 110
13 16 195
13 22 43
13 27 135
13 32 59
14 3 77
14 10 38
14 17 197
14 23 196
14 25 90
14 29 48
15 4 112
15 7 163
15 11 49
15 18 30
15 20 152
15 24 158
15 26 45
15 30 171
15 34 69
16 1 193
16 13 103
16 22 124

```
16 27 39
16 32 66
17 3 68
17 10 86
17 14 112
17 23 177
17 25 140
17 29 98
18 4 175
18 7 33
18 11 140
18 15 192
18 20 102
18 24 165
18 26 12
18 30 69
18 34 59
19 5 6
19 8 183
19 12 168
19 21 109
19 31 6
20 4 186
20 7 172
20 11 162
20 15 31
20 18 55
20 24 52
20 26 61
20 30 18
20 34 193
21 5 158
21 8 54
21 12 85
21 19 109
21 31 114
22 1 176
22 13 13
22 16 96
22 27 144
22 32 151
23 3 58
23 10 28
23 14 159
23 17 100
23 25 49
23 29 3
24 4 45
24 7 7
24 11 22
24 15 169
24 18 111
24 20 82
24 26 25
24 30 162
24 34 50
25 3 173
25 10 165
25 14 12
25 17 4
25 23 112
25 29 132
26 4 65
26 7 60
26 11 141
26 15 26
26 18 16
```

```
26 20 41
26 24 100
26 30 167
26 34 90
27 1 191
27 13 22
27 16 110
27 22 30
27 32 112
28 2 84
28 6 109
28 33 164
29 3 39
29 10 88
29 14 107
29 17 190
29 23 145
29 25 31
30 4 64
30 7 40
30 11 73
30 15 193
30 18 103
30 20 185
30 24 12
30 26 41
30 34 197
31 5 20
31 8 158
31 12 145
31 19 199
31 21 61
32 1 160
32 13 182
32 16 180
32 22 117
32 27 193
33 2 188
33 6 101
33 28 100
34 4 144
34 7 5
34 11 84
34 15 118
34 18 195
34 20 39
34 24 129
34 26 45
34 30 155

//---------------------------------------- rflowshop.cpp ------------------------------
// Program Description:
//     Generates the prb files for the reentrant flow shop problems for thesis
//     Works for all test problems of Demirkol and Uzsoy.

#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <fstream.h>
#include <ctype.h>
#include <iomanip.h>
#include <stdio.h>
using namespace std;

#define SIZE 50
#define DEBUG TRUE
```

```
struct Jobstr       // structure to hold the job details
{
  int num;          // store job number
  int arrival;      // arrival time of the job
  int duedate;      // due date of the job
  int seqarray[SIZE+1];  // job sequence or route
  int ptimearray[SIZE+1]; // proessing time array
};

char outpath[20]="rfsinst/";     // output file path
char ofile[40]="";                 // output file
char inpath[20]="uzoyprobs/";     // input file path
char ifile[40]="";                 // input file

FILE *fin, *fout;   // input and output file pointer

int main(int argc, char *argv[])   // allows to pass the arguments by command line
{
  int njob, nmach, nprod, nopers;
  int i, j, k;
  int dloopsize;
  int tracker;
  int rcount, ptcount, tcount, nprocess;
  int cumptime;
  float ops, loops, setupsevfact, setupdeviation;

  if (argc < 1)  // missing input and output file name?
  {
    cout <<" Usage: rflowshop <input filename> (.dat extension assumed)" << endl;
    exit(1);
  }

  if (!strcmp(argv[1], "-?"))  // user want to know how to use?
  {
    cout <<" USAGE: rflowshop <input filename> (.dat extension assumed)" << endl;
    exit(1);
  }

  sprintf(ifile, "%s%s.dat", inpath, argv[1]);
  if((fin=fopen(ifile,"r"))==NULL)  // can input file be opened?
  {
    cout << "Input file " << ifile << " opening failed." << endl;
    exit(1);
  }
  else
    cout << ifile << " opened successfully for reading." << endl;

  sprintf(ofile, "%s%s.prb", outpath, argv[1]);
  if((fout=fopen(ofile,"w"))==NULL)  // can output file be opened
  {
    cout << "Output file " << ofile << " opening failed." << endl;
    exit(1);
  }
  else
    cout << ofile << " opened successfully for writing." << endl;

  fprintf(fout, "%s\n", argv[1]);
  fscanf(fin, "%d %d %d", &nopers, &njob, &nmach);
  nprod = njob;  // each job is a product

  fprintf(fout, "%d %d %d\n", njob, nprod, nmach);

  int *resourcearray = new int[nmach+1];  // resource array - units of resource
  for(i=1;i<=nmach;i++)
    resourcearray[i] = 1;    // how many units  - currently 1 for flowshop

/*
```

```
 for(i=1;i<=nmach;i++)   // for each resource
 {
  fprintf(fout, "%d\n", resourcearray[i]);
  for(j=1;j<=nprod;j++)  // for each product
  {
    for(k=1;k<=nprod;k++)
      fprintf(fout, "%d ", 0);   // no sequence dependent setup times
    fprintf(fout, "\n");;
  }  // end of for loop (variable - j)
 } // end of for loop (variable - i)

 dloopsize = nmach*2;  // route information - (machine process-time) pair
 tcount = 0;           // initialize task count
 Jobstr *jobarray[njob+1];     // Array of structure for jobs
 for(i=1;i<=njob;i++)
 {
   jobarray[i] = new Jobstr;     // New instance of structure - dynamic allocation
   cumptime = 0;       // cumulative processing time
   jobarray[i]->arrival = 0;  // all jobs avialable at time 0
   jobarray[i]->num = i;
   tracker = 1;          // tracking variable for route and process time info
   rcount = 1;           // route index count
   ptcount = 1;          // processing time index cont
   for(j=1; j<=dloopsize; j++)
   {
     if((tracker % 2) != 0)  // if odd number
     {
        fscanf(fin, "%d", &jobarray[i]->seqarray[rcount]);
        rcount++;
        tcount++;
     }
     else                 // even number
     {
        fscanf(fin, "%d", &jobarray[i]->ptimearray[ptcount]);
        cumptime = cumptime + jobarray[i]->ptimearray[ptcount];
        ptcount++;
     }
     tracker++;
   }  // end of for loop - j
   jobarray[i]->duedate = cumptime;
   fscanf(fin, "\n");
 } // end of for loop - i

 cout << "Finished reading the data from " << ifile <<endl;

#if DEBUG
 cout << "no. of tasks :" << tcount << endl;
 for(i=1;i<=njob;i++)
 {
   cout << "Job :" << jobarray[i]->num << " Arrival :" << jobarray[i]->arrival << "
   Due Date :" << jobarray[i]->duedate << endl;
   for(j=1; j<=nmach; j++)
     cout << jobarray[i]->seqarray[j] << " " << jobarray[i]->ptimearray[j] << " ";
   cout << endl;
 }
#endif

 fprintf(fout, "%d\n", tcount);

 for(i=1;i<=njob;i++)   // machine # and process time for each job
 {
   fprintf(fout, "%d\n", jobarray[i]->num);   // write product number
   fprintf(fout, "%d\n", nmach);   // visits all machines (no. process = nmach)
   for(j=1;j<=nmach;j++)
     fprintf(fout, "%d %d ", jobarray[i]->seqarray[j], jobarray[i]->ptimearray[j]);
   fprintf(fout, "\n");
 } // end of for loop (variable - i)
```

```
 for(i=1;i<=njob;i++)   //arrival time for each job
    fprintf(fout, "%d\n", jobarray[i]->arrival);

 for(i=1;i<=njob;i++)
    fprintf(fout, "%d\n", jobarray[i]->duedate);

 ops = (float) nmach;
 loops = 1.0;
 fprintf(fout, "%f %f\n", ops, loops);   // average operations and loops

 setupsevfact = 0.0;
 setupdeviation = 0.0;
 fprintf(fout, "%f %f\n", setupsevfact, setupdeviation);

*/
 cout << "Created " << ofile << " in simulation readable format." << endl;

 fclose(fin);   // close input file
 fclose(fout);  // close output file

 return 1;
}

/* ------------------------------ anlyz.c ----------------------------
  Description:
    Functions in this file reads a file (*.run) and builds the
      a database (db .. *.dat files) which is used to anlayze relative
      performance of algorithms
  Revisions:
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <string.h>
#include <ctype.h>

#include "general.h"

/*  #define DEBUG */
 #define ERROR -1
 #define OK 0

 #define MAX_P 100   /* max problems considered */
 typedef struct pinfo {     /* problem table information */
   char pid[11];      /* problem id */
   int i;             /* problem index to run array */
   float zmax;        /* maximum z ( linear-quadratic ) for any algorithm */
   float zmin;
   float tmax;        /* max ( iterbest/niter ) over all algorithms */
   float tmin;
 } PROB;
 struct pinfo *ptbl[ MAX_P * 2 ];      /* problem info table */
 struct pinfo *prob;
 int nprob;                            /* number of problems */

 #define MAX_A 50
 typedef struct ainfo {     /* Algorithm table information */
   char aid[8];       /* algorithm id */
   int j;             /* algorithm index to run array */
   int np;            /* number of problems */
 } ALG;
 struct ainfo *atbl[ MAX_A * 2 ];      /* algorithm table */
 struct ainfo *alg;
 int nalg;                             /* number of algorithms */
```

```
 struct runinfo {   /* Problem-algorithm run info. SUMMED OVER ALL REPS */
    long niter;         /* total iterations run */
    float time;        /* search time in seconds */
    long iterbest;      /* iteration when best solution found */
    float tbest;         /* Time best solution found */
    float tb;          /* sum ( over reps ) time ( in min.) best soln. found */
    double sstb;       /* sum ( over reps ) of squared tb */
    float mintb;       /* minimum tbest (over reps ) */
   float maxtb;
   float z1best;      /* linear objective ( assign. value ) */
   float scxx1;       /* soft conflicts */
   double sszb;       /* sum (over reps ) of squared zbest (linear-quad) */
   float minzb;       /* minimum zbest (over reps ) */
   float maxzb;
   float scxx2;       /* hard conflicts */
   double ssc2;       /* sum of squared hard conflicts */
   float dindx;       /* diversification index */
   int itercxx2;      /* Iteration when min (best) sum{cxx2} found */
   float tf;          /* sum of first feasible soln times ( in minutes ) */
   double sstf;       /* sum of squared tf ( time most feas first ) */
   float zfeas;       /* Z value ( linear - quadratic ) at itercxx2 */
   double sszf;        /* ssq for zfeas */
   int nreps;         /* Number of replications ( prob-algorithm ) */
   int nfreps;        /* No. of feasible reps */
 };
 struct runinfo * runs [MAX_P] [MAX_A];

 struct pinfo *MakeProb( char *pid );
 struct ainfo *MakeAlg( char *aid );
 struct runinfo *MakeRun();
 int InsrtTab( void *rec, void *tbl[], int *r );
 void *SrchTab( char *key, void *tbl[], int nrec );
 void LoadRuns( char *runfnam );
 void AlgAnal();

 void DAnal();
 void Stats( float nobs, float sum, double ssq, float *avg, float *sdv );

 int CmpAid( ALG ** a1, ALG ** a2 );
 int CmpPid( PROB **p1, PROB **p2 );

 char EOFld( char ch );
 char Empty( char *buf );
 void ClrBuf( char *inbuf, int lstchr);

 #define MAXINRLEN 200
 char inbuf[MAXINRLEN + 1];      /* input buffer .. */
 char debug = 1;                 /* debug output flag */
 char dbpath[64]="";             /* directory path for database files */

 #define Max( A, B ) ((A) > (B) ? (A) : (B))
 #define Min( A, B ) ((A) < (B) ? (A) : (B))

main( int argc, char *argv[] )
{
 extern int rlen();
 int f = 1, i, j;
 int c;
 int detail = FALSE;          /* Detailed Run analysis Flag */
 char chr, runfnam[14]="";

  while ( --argc > 0 ) {
    if ( (*++argv) [0] == '-' ) {
      while ( (chr = *++argv[0]) != '\0' ) {
        switch (chr) {
          case 'd':         /* produce detailed run analysis */
```

```
            detail = TRUE;
            break;
          default:
            printf( "** WARNING - unknown option - %c **\n", chr );
            break;
        } /* switch */
      }
      --argc;
    }
    else {
      printf("\n ** ERROR - Usage: anlyz [-d]");
      exit(1);
    }
  }

  for ( i=0; i<MAX_P; i++ )
    for ( j=0; j<MAX_A; j++ ) runs[i][j] = NULL;

/* == Do Analysis == */

  do {
    printf( "\nInput -> Run File Name /e to finish )? " );
    gets( runfnam );
    if ( *runfnam == '/' && tolower( runfnam[1] ) == 'e' ) break;

    LoadRuns( runfnam );

     qsort( ( ALG **) atbl, (size_t) nalg, sizeof( ALG * ),
                    ( int (*) (const void *, const void *) ) CmpAid );
     AlgAnal();

     if ( detail ) {
        qsort( ( PROB **) ptbl, (size_t) nprob, sizeof( PROB * ),
               ( int (*) (const void *, const void *) ) CmpPid );
        DAnal();
    }

    fflush(stdout);
    for ( i=0; i<nprob; i++ ) {
      free( ptbl[i] );
      for ( j=0; j<nalg; j++ ) {
        free( runs [i] [j] );
        runs [i] [j] = NULL;
      }
    }
    for ( j=0; j<nalg; j++ ) free( atbl[j] );

  } while (1);

} /* end main() */


/* ------------------------------------------------------------ LoadRuns ()
  Loads algorithm performance information for several runs ..
*/
void LoadRuns( char *runfnam )
{
 FILE *fin;
 char fnm[80]="";            /* complete file name to input */
 time_t curtime;             /* stores calendar time for output */

 struct runinfo *run;
 char fld[50];
 char *cp, *seps = ",";
 char run_pid[15], run_aid[15];

 int i, j, r, p, a, c, ntbuelst;
```

```
float runz, runtbest;

long niter, ibest, ifeas;
float rtime, tb, z1b, c1b, c2b, tf, zf;
float dindx;

 curtime = time(NULL);
 sprintf( fnm, "%s%s", dbpath, runfnam );      /* PROBLEM input file */
 if (( fin = fopen (fnm,"r")) == NULL) {
   printf("** ERROR - LoadRuns() - Could not open file %s \n", fnm);
    rewind( fin );
   exit(1);
 }
 printf( "\f    ANLYZ.C Run, File:  %s  -  %s\n\n", fnm, ctime(&curtime) );

/* === Load the RUNINFO table  */
 r = nprob = nalg = 0;
 ClrBuf( inbuf, MAXINRLEN );
 while ( fgets( inbuf, MAXINRLEN, fin) && ! EOFld( inbuf[0] ) ) {

  /* Break into fields. */
   strtok( inbuf, seps );       /* ignore ist field */
    strcpy( run_aid, strtok( NULL, seps ) );

    /* alternate code: elm 8/95
    strcpy( fld, strtok( NULL, seps ) );
    if ( ( cp = strrchr( fld, '/' ) ) == NULL ) {
       if ( ( cp = strrchr( fld, '\\' ) ) == NULL )
         cp = fld;
    }
    strcpy( run_aid, (cp+1) );     last directory in cwd */

    if ( ( alg = SrchTab( run_aid, atbl, nalg ) )  ==  NULL ) {
       alg = MakeAlg( run_aid );       /* create algorithm and put in table */
    }

    strcpy( run_pid, strtok( NULL, seps ) );
    strtok( NULL, seps );   /* ignore parameter set no. */
    /*strncat( run_pid, strtok( NULL,seps ), 6 ); */
    /*strtok( NULL, seps );      skip path */
    if ( ( prob = SrchTab( run_pid, ptbl, nprob ) )  ==  NULL ) {
       prob = MakeProb( run_pid );       /* create problem and put in table */
    }

    if ( runs[prob->i] [alg->j] == NULL )
       runs [prob->i] [alg->j] = MakeRun();

    run = runs [prob->i] [alg->j];   /* put run in table */

    strtok( NULL, seps );                        /* startopt */
   /* Following is summed over Replicates */
    niter = atol( strtok( NULL, seps ) );        /* iteration */
    run->niter += niter;
    rtime = atof( strtok( NULL, seps ) );  /* run srch_time (secs.) */
    run->time += rtime;

    ibest = atol( strtok( NULL, seps ) );
    run->iterbest += ibest;

    tb = atof( strtok( NULL, seps ) );   /* Time to best val. (minutes)*/
    if ( tb < 0 )   /* overflow ?? */
       tb = rtime * ibest / niter / 60.0;
    run->tbest = tb;
    run->tb += tb;
    run->sstb += tb * tb;
    if ( run->mintb > tb ) run->mintb = tb;
    if ( run->maxtb < tb ) run->maxtb = tb;
```

```
      strtok( NULL, seps );          /* initial proximity wt. */
      z1b = atof( strtok( NULL, seps ) );  /* best proximity wt. */
      c1b = 0;                        /* no conflicts */
      run->z1best += z1b;
      run->scxx1 += c1b;
      run->sszb += ( z1b - c1b ) * ( z1b - c1b );
      if ( run->minzb > z1b-c1b ) run->minzb = z1b - c1b;
      if ( run->maxzb < z1b-c1b ) run->maxzb = z1b - c1b;

      c2b = 0;
      run->scxx2 += c2b;
      run->ssc2 += c2b * c2b;

      dindx = 0;
      run->dindx += dindx;

      ifeas = 0;
      run->itercxx2 += ifeas;
      tf = 0;
      run->tf += tf;
      run->sstf += tf * tf;

      zf = 0;
      run->zfeas += zf;
      run->sszf += zf * zf;

      run->nreps++;
      if ( c2b == 0.0 ) run->nfreps++;

      if ( run->nreps == 1 ) alg->np++;  /* count each problem once only */


#ifdef DEBUG
      printf("\n * r: %d *\n", r );
#endif

      ClrBuf( inbuf, MAXINRLEN );
      r++;
   } /* next run */

  /* Now, Get mins and maxs (averaged over reps ) for each problem */
   for ( p=0; p<nprob; p++ ) {
      prob = ptbl[p];
     for ( a=0; a<nalg; a++ ) {
       alg = atbl[a];
       run = runs [prob->i] [alg->j];
       if ( run == NULL ) continue;

       runz = ( run->z1best - run->scxx1 ) / run->nreps;   /* average .. */
       if ( runz > prob->zmax ) prob->zmax = runz;
       if ( runz < prob->zmin ) prob->zmin = runz;

       runtbest = (float ) run->tb / run->nreps;  /* avg. too */
       if ( runtbest > prob->tmax ) prob->tmax = runtbest;
       if ( runtbest < prob->tmin ) prob->tmin = runtbest;
     }
   }
   fclose(fin);
} /* end LoadRuns() */


/* ------------------------------------------------------------- AlgAnal()
   Analyzes average run results by algorithm
    runs.. stored in the runs[][] array

   NOTES:
```

```
   1. Relative measures (esp. time) assume equal run TIMES
      for each algorithm
   2. Averages for ratios (e.g. iterbest/niter) assumes that
        each run replication has nearly the same approx. iterations ..
     => run replicates are over starting pts. and/or sampling streams
        but NOT problem instances which may affect total run iterations
  3-1-91, created ... elm
*/
void AlgAnal()
{
 int a, p, i, j, np;
 float hcon, sum_hcon[MAX_A], min_hcon[MAX_A], max_hcon[MAX_A];
 float zfeas, sum_zfeas[MAX_A], min_zfeas[MAX_A], max_zfeas[MAX_A];
 float tcxx2, sum_tcxx2[MAX_A], min_tcxx2[MAX_A], max_tcxx2[MAX_A];

 float zbest, sum_zbest[MAX_A], min_zbest[MAX_A], max_zbest[MAX_A];
 float zdiff, sum_zdiff[MAX_A], min_zdiff[MAX_A], max_zdiff[MAX_A];

 float tbest, sum_tbest[MAX_A], min_tbest[MAX_A], max_tbest[MAX_A];
 float tdiff, sum_tdiff[MAX_A], min_tdiff[MAX_A], max_tdiff[MAX_A];

 float ipsec, sum_ipsec[MAX_A], min_ipsec[MAX_A], max_ipsec[MAX_A];
 float dindx, sum_dindx[MAX_A], min_dindx[MAX_A], max_dindx[MAX_A];
 float zratio, zrank, avgt, trank;
 float avghcon, avgtc, avgzf, avgd;

 struct runinfo *run;
 struct pinfo *prob;
 struct ainfo *alg;

  printf( "                          Min.                     Tbest=            (Zrank+\n" );
  printf( "             Iter/ Dvrs  Hard   ZF/  IterF/  ZB/  IterB/             Trank)/\n" );
  printf( "     Alg. Np  Sec  Indx  Con.  Zmaxp Niter  Zmaxp Niter  Zrank Trank    2   \n" );
  printf( "-------------------------------------------------------------------------------\n" );

  for ( a=0; a<nalg; a++ ) {    /* for each algorithm */
    alg = atbl[a];
    j = alg->j;
    printf( "%8s", alg->aid );
    sum_hcon[j] = sum_zfeas[j] = sum_tcxx2[j] = 0;
    sum_zbest[j] = sum_zdiff[j] = sum_tbest[j] = sum_tdiff[j] = 0.0;
    sum_ipsec[j] = sum_dindx[j] = 0.0;
    min_hcon[j] = min_zfeas[j] = min_tcxx2[j] = 1e20;
    min_zbest[j] = min_zdiff[j] = min_tbest[j] = min_tdiff[j] = 1e20;
    min_ipsec[j] = min_dindx[j] = 1e20;
    max_hcon[j] = max_zfeas[j] = max_tcxx2[j] = -1e20;
    max_zbest[j] = max_zdiff[j] = max_tbest[j] = max_tdiff[j] = -1e20;
    max_ipsec[j] = max_dindx[j] = -1e20;

    for ( p=0; p<nprob; p++ ) {               /* sum for all problems */
      prob = ptbl[p];
      i = prob->i;
      if ( runs[i][j] == NULL ) continue;
      run = runs[i][j];

        ipsec = (float) run->niter/ run->time;
      sum_ipsec[j] += ipsec;
      min_ipsec[j] = Min( min_ipsec[j], ipsec );
      max_ipsec[j] = Max( max_ipsec[j], ipsec );

      dindx = run->dindx / run->nreps;
      sum_dindx[j] += dindx;
      min_dindx[j] = Min( min_dindx[j], dindx );
      max_dindx[j] = Max( max_dindx[j], dindx );

      hcon = run->scxx2 / run->nreps;
      sum_hcon[j] += hcon;
```

```
      min_hcon[j] = Min( min_hcon[j], hcon );
      max_hcon[j] = Max( max_hcon[j], hcon );

      zfeas =  run->zfeas / run->nreps;
      sum_zfeas[j] += zfeas / prob->zmax;
      min_zfeas[j] = Min( min_zfeas[j], zfeas / prob->zmax );
      max_zfeas[j] = Max( max_zfeas[j], zfeas / prob->zmax );

      tcxx2 = (float) run->itercxx2 / (float) run->niter;
      sum_tcxx2[j] += tcxx2;
      min_tcxx2[j] = Min( min_tcxx2[j], tcxx2 );
      max_tcxx2[j] = Max( max_tcxx2[j], tcxx2 );

      zbest = ( run->z1best - run->scxx1 ) / run->nreps;
      sum_zbest[j] += zbest / prob->zmax;
      min_zbest[j] = Min( min_zbest[j], zbest / prob->zmax );
      max_zbest[j] = Max( max_zbest[j], zbest / prob->zmax );

      zdiff = 0.0;
      if ( prob->zmax > prob->zmin )
        zdiff = ( zbest - prob->zmin ) / ( prob->zmax - prob->zmin );
      sum_zdiff[j] += zdiff;
      min_zdiff[j] = Min( min_zdiff[j], zdiff );
      max_zdiff[j] = Max( max_zdiff[j], zdiff );

        tbest = (float) run->tb / (float) run->nreps;
        sum_tbest[j] += 1.0 - tbest / prob->tmax;
        min_tbest[j] = Min( min_tbest[j], 1.0 - tbest / prob->tmax );
        max_tbest[j] = Max( max_tbest[j], 1.0 - tbest / prob->tmax );

      tdiff = 0.0;
      if ( prob->tmax > prob->tmin )
        tdiff = ( tbest - prob->tmin ) / ( prob->tmax - prob->tmin );
      sum_tdiff[j] += tdiff;
      min_tdiff[j] = Min( min_tdiff[j], tdiff );
      max_tdiff[j] = Max( max_tdiff[j], tdiff );

    }  /* next i */

    np = alg->np;
    avghcon = sum_hcon[j] / np;  /* avg. hard conflicts */
    avgzf = sum_zfeas[j] / np;   /* avg. z value when min conflicts found */
    avgtc = sum_tcxx2[j] / np;   /* avg. % time till feasibility */
    zratio = sum_zbest[j] / np;  /* avg ratio */
    zrank = sum_zdiff[j] / np;
    avgt = sum_tbest[j] / np;
    trank = 1 - sum_tdiff[j] / np;
    avgd = sum_dindx[j] / np;        /* avg. diversification index */

    printf( " %3d %4.2f %4.3f", np, sum_ipsec[j]/np, avgd );
    printf( " %5.1f  %4.3f %4.3f  %4.3f %4.3f  %4.3f %4.3f  %4.2f\n",
   avghcon, avgzf, avgtc, zratio, avgt, zrank, trank, (zrank + trank ) / 2 );
    printf( "             %4.2f %4.3f", min_ipsec[j], min_dindx[j] );
    printf( " %5.1f  %4.3f %4.3f  %4.3f %4.3f  %4.3f %4.3f\n",
    min_hcon[j], min_zfeas[j], min_tcxx2[j],
    min_zbest[j], min_tbest[j], min_zdiff[j], 1 - max_tdiff[j] );
    printf( "             %4.2f %4.3f", max_ipsec[j], max_dindx[j] );
    printf( " %5.1f  %4.3f %4.3f  %4.3f %4.3f  %4.3f %4.3f\n\n",
    max_hcon[j], max_zfeas[j], max_tcxx2[j],
    max_zbest[j], max_tbest[j], max_zdiff[j], 1 - min_tdiff[j] );

  } /* next j */

} /* end AlgAnal() */


/* ------------------------------------------------------------------ DAnal()
```

```
  Analyzes average results for each run ( across all replicates )
   runs.. stored in the runs[][] array
  Outputs a line for each run ( problem<->algorithm pair ) where runs
   may be replicated.
  Absolute solution quality and run time measures only are output here ..

  4-18-91, created ... elm
*/
void DAnal()
{
 #define BLKS_PER_PAGE 8
 struct runinfo *run;
 struct pinfo *prob;
 struct ainfo *alg;

 int a, p, i, j;
 float c2_avg, zf_avg, tf_avg, zb_avg, tb_avg;
 float c2_sdv, zf_sdv, tf_sdv, zb_sdv, tb_sdv;
 float nn, ssx, avgi;

 int nreps_p, nfreps_p;                     /* Problem Stats */
 float sc2_p, szf_p, stf_p, minzb_p, maxzb_p, szb_p, mintb_p, maxtb_p, stb_p;
 double ssc2_p, sszf_p, sstf_p, sszb_p, sstb_p;



  for ( p=0; p<nprob; p++ ) {
     if ( p % BLKS_PER_PAGE == 0 ) {        /* New Page ? */
        printf( "\f\n\n" );
        printf( "                 No. Min. Hard \n" );
        printf( "    Prob        Feas Confilcts     Z_Feas     T_F (min.)
                                                    Z-Best              T-Best (minutes)\n" );
        printf( "     Alg.  Nrep Reps  Avg  Sdv    Avg  Sdv     Avg  Sdv
                                                   Min.   Max.   Avg.   Sdv    Min.  Max.  Avg.  Sdv\n" );
        printf( "-------------------------------------------------------------------
                                                  ---------------------------------------" );
     }

     prob = ptbl[p];
     i = prob->i;
    nreps_p = nfreps_p = 0;                  /* Initialize Problem Stats */
    minzb_p = mintb_p = 1e20;
    maxzb_p = maxtb_p = -1e20;
    sc2_p = szf_p = stf_p = szb_p = stb_p = 0;
    ssc2_p = sszf_p = sstf_p = sszb_p = sstb_p = 0;

    printf( "\n%10s\n", prob->pid );
    for ( a=0; a<nalg; a++ ) {
      alg = atbl[a];
      j = alg->j;
      run = runs [i] [j];
      if ( run == NULL ) continue;

      nn = run->nreps;
      Stats( nn, run->scxx2, run->ssc2, &c2_avg, &c2_sdv );
      Stats( nn, run->zfeas, run->sszf, &zf_avg, &zf_sdv );

/* -- Old Ratio Calculations . .... changed to times 4/22/91 .. elm
      tf_avg = (float) run->itercxx2 / (float) run->niter;
      avgi = ( (float) run->niter / nn );
      ssx = ( run->ssif /  ( avgi * avgi ) );
      tf_sdv = nn < 2  ?  0.0  :  ( ssx - nn*tf_avg*tf_avg ) / ( nn - 1 );
      tf_sdv = tf_sdv < 0.0  ?  0.0  :  sqrt( tf_sdv );
*/
      Stats( nn, run->tf, run->sstf, &tf_avg, &tf_sdv );
      Stats( nn, (run->z1best - run->scxx1), run->sszb, &zb_avg, &zb_sdv );
      Stats( nn, run->tb, run->sstb, &tb_avg, &tb_sdv );
```

```
     /* Output Run Stats Line */
      printf( "  %8s %4d %4d ", alg->aid, run->nreps, run->nfreps );
      printf( "%4.1f %4.1f  ", c2_avg, c2_sdv );
      printf( "%6.1f %5.1f  %5.2f %5.2f  ", zf_avg, zf_sdv, tf_avg, tf_sdv );
      printf( "%6.1f %6.1f %6.1f %5.1f  ",
              run->minzb, run->maxzb, zb_avg, zb_sdv );
      printf( "%5.2f %5.2f %5.2f %5.2f\n",
              run->mintb, run->maxtb, tb_avg, tb_sdv );

     /* Accumulate problem stats */
      nreps_p += run->nreps;
      nfreps_p += run->nfreps;
      sc2_p += run->scxx2;
      ssc2_p += run->ssc2;
      szf_p += run->zfeas;
      sszf_p += run->sszf;
      stf_p += run->tf;
      sstf_p += run->sstf;
      if ( run->minzb < minzb_p ) minzb_p = run->minzb;
      if ( run->maxzb > maxzb_p ) maxzb_p = run->maxzb;
      szb_p += ( run->z1best - run->scxx1 );
      sszb_p += run->sszb;
      if ( run->mintb < mintb_p ) mintb_p = run->mintb;
      if ( run->maxtb > maxtb_p ) maxtb_p = run->maxtb;
      stb_p += run->tb;
      sstb_p += run->sstb;

    }
   /* Output Problem Cum Stats */
    nn = nreps_p;
    Stats( nn, sc2_p, ssc2_p, &c2_avg, &c2_sdv );
    Stats( nn, szf_p, sszf_p, &zf_avg, &zf_sdv );
    Stats( nn, stf_p, sstf_p, &tf_avg, &tf_sdv );
    Stats( nn, szb_p, sszb_p, &zb_avg, &zb_sdv );
    Stats( nn, stb_p, sstb_p, &tb_avg, &tb_sdv );

    printf( "           ---- ---- ---- ----  ------ -----  ----- -----  ------ ------ ------
                                              -----  ----- ----- ----- -----\n" );
    printf( "  %8s %4d %4d ", "     All", nreps_p, nfreps_p );
    printf( "%4.1f %4.1f  ", c2_avg, c2_sdv );
    printf( "%6.1f %5.1f  %5.2f %5.2f  ", zf_avg, zf_sdv, tf_avg, tf_sdv );
    printf( "%6.1f %6.1f %6.1f %5.1f  ",
            minzb_p, maxzb_p, zb_avg, zb_sdv );
    printf( "%5.2f %5.2f %5.2f %5.2f\n",
              mintb_p, maxtb_p, tb_avg, tb_sdv );

  }

} /* end DAnal() */


/* ---------------------------------------------------------------- Stats()
  Computes an avg and Standard deviation from sumof squares
*/
void Stats( float nobs, float sum, double ssq, float *avg, float *sdv )
{
float xavg, xsdv;
  xavg = sum / nobs;
  xsdv = nobs < 2  ?  0.0  : ( ssq - nobs * xavg * xavg ) / ( nobs - 1);
  xsdv = xsdv < 0.0  ?  0.0  :  sqrt( xsdv );
  *avg = xavg;
  *sdv = xsdv;
} /* end Stats() */


/* ---------------------------------------------------------------- CmpAid()
```

```
 Compares two algorithm id's .. from internal algorithm table ..
  returns greater than (1), less than (-1), or equal to (0).
  This function is called by qsort and bsearch.
*/
int CmpAid( ALG ** a1, ALG ** a2 )
{
  return strcmp( (*a1)->aid, (*a2)->aid );
}


/* ------------------------------------------------------------- CmpPid()
 Compares two problem id's .. from internal problem table ..
  returns greater than (1), less than (-1), or equal to (0).
  This function is called by qsort and bsearch.
*/
int CmpPid( PROB ** p1, PROB ** p2 )
{
  return strcmp( (*p1)->pid, (*p2)->pid );
}


/* ------------------------------------------------------------ MakeProb()
  Creates and initializes a Problem structure
  Returns address of the new structure .. NULL if a problem ..
 03-01-91, .. Created ...  elm
*/
struct pinfo *MakeProb( char *pid )
{
 struct pinfo *temp;

 if ( nprob >= MAX_P ) return NULL;

 temp = ( struct pinfo * ) malloc( sizeof( struct pinfo ) );
 if ( temp == NULL ) {
   printf ( "*** ERROR - MakeProb() - out of memory ***");
   return NULL;
 }
 strcpy( temp->pid, pid );
 temp->i = nprob;
 temp->zmax = -1e20;
 temp->zmin = +1e20;
 temp->tmax = -1e20;
 temp->tmin = +1e20;
 InsrtTab( temp, ptbl, &nprob );
 ++nprob;

 return temp;
} /* end MakeProb() */

/* ------------------------------------------------------------- MakeAlg()
  Creates and initializes a Algorithm structure
  Returns address of the new structure .. NULL if a problem ..
 03-01-91, .. Created ...  elm
*/
struct ainfo *MakeAlg( char *aid )
{
 struct ainfo *temp;

 if ( nalg >= MAX_A ) return NULL;

 temp = ( struct ainfo * ) malloc( sizeof( struct ainfo ) );
 if ( temp == NULL ) {
   printf ( "*** ERROR - MakeAlg() - out of memory ***");
   return NULL;
 }
 strcpy( temp->aid, aid );
 temp->j = nalg;
```

```
 temp->np = 0;
 InsrtTab( temp, atbl, &nalg );
 ++nalg;

 return temp;
} /* end MakeAlg() */


/* ------------------------------------------------------------ MakeRun()
  Creates and initializes a run (problem-algorithm data) structure
  Returns address of the new structure .. NULL if a problem ..
   4/17/91, .. Created ...  elm
*/
struct runinfo *MakeRun()
{
 struct runinfo *run;

  run = ( struct runinfo * ) malloc( sizeof( struct runinfo ) );
  if ( run == NULL ) {
    printf ( "*** ERROR - MakeRun() - out of memory ***");
    return NULL;
  }

 /* Initialize to zero as all are sums across Replications */

  run->niter = 0;       /* total iterations run */
  run->time = 0;        /* search time in seconds */
  run->iterbest = 0;    /* iteration when best solution found */
  run->tb = 0.0;        /* sum of time ( in min. ) when best z found */
  run->sstb = 0.0;      /* sum ( over reps ) of squared tbest */
  run->mintb = 1e20;
  run->maxtb = -1e20;
  run->z1best = 0.0;    /* linear objective ( assign. value ) */
  run->scxx1 = 0.0;     /* soft conflicts */
  run->sszb = 0.0;      /* sum (over reps ) of squared zbest (linear-quad) */
  run->minzb = 1e20;
  run->maxzb = -1e20;
  run->scxx2 = 0.0;     /* hard conflicts */
  run->ssc2 = 0.0;      /* sum of squared hard conflicts */
  run->dindx = 0.0;     /* diversification index */
  run->itercxx2 = 0;    /* Iteration when min (best) sum{cxx2} found */
  run->tf = 0.0;        /* Time ( in min. ) of itercxx2 */
  run->sstf = 0.0;      /* sum of squared tf ( time most  feas ) */
  run->zfeas = 0.0;     /* Z value ( linear - quadratic ) at itercxx2 */
  run->sszf = 0.0;      /* ssq for zfeas */
  run->nreps = 0;       /* Number of replications ( prob-algorithm ) */
  run->nfreps = 0;      /* No. of feasible reps .. */

  return run;
} /* end MakeRun */


/* ------------------------------------------------------------ InsrtTab()
  Inserts records into a table of records whose first
 field is a string key representation
  Returns OK or ERROR, depending on results ..  updates nrec ..
 04-17-90, .. Created for timetable dbase ...  elm
*/
int InsrtTab( void *rec, void *tbl[], int *r )
{
  tbl [*r] = rec;
} /* end InsrtTab() */


/* ------------------------------------------------------------ SrchTab()
  Performs a linear search of a table of records whose first
 field is a string key representation
```

```
  Returns a pointer to the found record or NULL if not found
  04-17-90, .. Created for timetable dbase ...  elm
*/
void *SrchTab( char *key, void *tbl[], int nrec )
{
 int i;

  for ( i=0; i<nrec; i++ )
    if ( !strcmp( key, tbl[i] ) ) return tbl[i];

  return NULL;
} /* end SrchTab() */


/* ======================== Misc. Utilities ============================ */

/* ------------------------------------------------------------ EOFld ()
  Function to check a character for end of field membership
              ( \r, \n, or \0 ..)
  04-17-90, .. Created for timetable dbase ...  elm
*/
char EOFld( char ch )
{
  if ( ch == '\0' || ch == '\r' || ch == '\n' ) return TRUE;
  else return FALSE;
}

/* ------------------------------------------------------------ Empty ()
  Function to check a char array for all blanks
  04-18-90, .. Created for timetable dbase ...  elm
*/
char Empty( char *buf )
{
int i;
  for (i=0; buf[i] != '\0'; i++) if (buf[i] != ' ') return FALSE;
  return TRUE;
}

/* ------------------------------------------------------------ ClrBuf ()
  Function to blank to a buffer from char 0 throu lstchr (inclusive)
  04-17-90, .. Created for timetable dbase ...  elm
*/
void ClrBuf( char *inbuf, int lstchr)
{
int i;
  for (i=0; i<=lstchr; i++) inbuf[i] = ' ';
} /* end ClrBuf() */
```

The experimental design and the results for RFJSSDS problems are summarized below. The design shows how each instances were generated, by keeping the factors at various treatment levels. Following the experimental design, the details of each RFJSSDS problem set from the body of the thesis is summarized.

```
//======================== Experimental design ========================

Varying Factors                    Design Grid
Sl. No  Name    label   -   +
1   jobs        A   5   10      Num Std Order   A   B   C   D   S   P   T
2   operations  B   2   4       1   (I) -   -   -   -   -   -   -
3   loops       C   2   4       2   a   +   -   -   -   -   -   -
4   load factor D   0.3 0.7     3   b   -   +   -   -   -   -   -
5   capacity    E   1   3       4   ab  +   +   -   -   -   -   -
6   mean p time F   20  40      5   c   -   -   +   -   -   -   -
7   tardy fact G   0.3 0.6      6   ac  +   -   +   -   -   -   -
                                7   bc  -   +   +   -   -   -   -
Replications (seeds)            8   abc +   +   +   -   -   -   -
                                9   d   -   -   -   +   -   -   -
1   12345                       10  ad  +   -   -   +   -   -   -
2   98754                       11  bd  -   +   -   +   -   -   -
3   35689                       12  abd +   +   -   +   -   -   -
4   23896                       13  cd  -   -   +   +   -   -   -
5   45698                       14  acd +   -   +   +   -   -   -
                                15  bcd -   +   +   +   -   -   -
Fixed Factors                   16  abcd    +   +   +   +   -   -   -
Sl. No  Name                    17  e   -   -   -   -   +   -   -
1   setup severity  0.2         18  ae  +   -   -   -   +   -   -
2   setup deviation 0.1         19  be  -   +   -   -   +   -   -
3   prod fact deviation 0.2     20  abe +   +   -   -   +   -   -
4   proc fact deviation 0.1     21  ce  -   -   +   -   +   -   -
5   ps factor   0.1, 0.2        22  ace +   -   +   -   +   -   -
6   prod mix array  0, 0        23  bce -   +   +   -   +   -   -
7   number of products  2   3   24  abce    +   +   +   -   +   -   -
                                25  de  -   -   -   +   +   -   -
                                 26  ade +   -   -   +   +   -   -
                                27  bde -   +   -   +   +   -   -
                          28  abde    +   +   -   +   +   -   -
                          29  cde -   -   +   +   +   -   -
                          30  acde    +   -   +   +   +   -   -
                          31  bcde    -   +   +   +   +   -   -
                          32  abcde   +   +   +   +   +   -   -
                          33  f   -   -   -   -   -   +   -
                          34  af  +   -   -   -   -   +   -
                          35  bf  -   +   -   -   -   +   -
                          36  abf +   +   -   -   -   +   -
                          37  cf  -   -   +   -   -   +   -
                          38  acf +   -   +   -   -   +   -
                          39  bcf -   +   +   -   -   +   -
                          40  abcf    +   +   +   -   -   +   -
                          41  df  -   -   -   +   -   +   -
                          42  adf +   -   -   +   -   +   -
                          43  bdf -   +   -   +   -   +   -
                          44  abdf    +   +   -   +   -   +   -
                          45  cdf -   -   +   +   -   +   -
                          46  acdf    +   -   +   +   -   +   -
                          47  bcdf    -   +   +   +   -   +   -
                          48  abcdf   +   +   +   +   -   +   -
                          49  ef  -   -   -   -   +   +   -
                          50  aef +   -   -   -   +   +   -
                          51  bef -   +   -   -   +   +   -
                          52  abef    +   +   -   -   +   +   -
                          53  cef -   -   +   -   +   +   -
                          54  acef    +   -   +   -   +   +   -
                          55  bcef    -   +   +   -   +   +   -
                          56  abcef   +   +   +   -   +   +   -
                          57  def -   -   -   +   +   +   -
                          58  adef    +   -   -   +   +   +   -
                          59  bdef    -   +   -   +   +   +   -
                          60  abdef   +   +   -   +   +   +   -
                          61  cdef    -   -   +   +   +   +   -
```

62	acdef	+	−	+	+	+	+	−
63	bcdef	−	+	+	+	+	+	−
64	abcdef	+	+	+	+	+	+	−
65	g	−	−	−	−	−	−	+
66	ag	+	−	−	−	−	−	+
67	bg	−	+	−	−	−	−	+
68	abg	+	+	−	−	−	−	+
69	cg	−	−	+	−	−	−	+
70	acg	+	−	+	−	−	−	+
71	bcg	−	+	+	−	−	−	+
72	abcg	+	+	+	−	−	−	+
73	dg	−	−	−	+	−	−	+
74	adg	+	−	−	+	−	−	+
75	bdg	−	+	−	+	−	−	+
76	abdg	+	+	−	+	−	−	+
77	cdg	−	−	+	+	−	−	+
78	acdg	+	−	+	+	−	−	+
79	bcdg	−	+	+	+	−	−	+
80	abcdg	+	+	+	+	−	−	+
81	eg	−	−	−	−	+	−	+
82	aeg	+	−	−	−	+	−	+
83	beg	−	+	−	−	+	−	+
84	abeg	+	+	−	−	+	−	+
85	ceg	−	−	+	−	+	−	+
86	aceg	+	−	+	−	+	−	+
87	bceg	−	+	+	−	+	−	+
88	abceg	+	+	+	−	+	−	+
89	deg	−	−	−	+	+	−	+
90	adeg	+	−	−	+	+	−	+
91	bdeg	−	+	−	+	+	−	+
92	abdeg	+	+	−	+	+	−	+
93	cdeg	−	−	+	+	+	−	+
94	acdeg	+	−	+	+	+	−	+
95	bcdeg	−	+	+	+	+	−	+
96	abcdeg	+	+	+	+	+	−	+
97	fg	−	−	−	−	+	+	
98	afg	+	−	−	−	+	+	
99	bfg	−	+	−	−	+	+	
100	abfg	+	+	−	−	−	+	+
101	cfg	−	+	−	−	+	+	
102	acfg	+	−	+	−	−	+	+
103	bcfg	−	+	+	−	−	+	+
104	abcfg	+	+	+	−	−	+	+
105	dfg	−	−	+	−	+	+	
106	adfg	+	−	−	+	−	+	+
107	bdfg	−	+	−	+	−	+	+
108	abdfg	+	+	−	+	−	+	+
109	cdfg	−	−	+	+	−	+	+
110	acdfg	+	−	+	+	−	+	+
111	bcdfg	−	+	+	+	−	+	+
112	abcdfg	+	+	+	+	−	+	+
113	efg	−	−	−	+	+	+	
114	aefg	+	−	−	−	+	+	+
115	befg	−	+	−	−	+	+	+
116	abefg	+	+	−	−	+	+	+
117	cefg	−	−	+	−	+	+	+
118	acefg	+	−	+	−	+	+	+
119	bcefg	−	+	+	−	+	+	+
120	abcefg	+	+	+	−	+	+	+
121	defg	−	−	−	+	+	+	+
122	adefg	+	−	−	+	+	+	+
123	bdefg	−	+	−	+	+	+	+
124	abdefg	+	+	−	+	+	+	+
125	cdefg	−	−	+	+	+	+	+
126	acdefg	+	−	+	+	+	+	+
127	bcdefg	−	+	+	+	+	+	+
128	abcdefg	+	+	+	+	+	+	+

Summary of actual values for for N=5, M=2, L=0.3 problems

C	S	P	T	Rep	LB	EDD	TIME	SPT	TIME	LSLK	TIME	BRSP	TIME	SBLIMS	TIME
2	1	20	0.3	1	152	175	0.032	171	0.032	181	0.044	165	0.14	162	0.41
2	1	20	0.3	2	148	173	0.034	167	0.034	178	0.048	160	0.15	158	0.37
2	1	20	0.3	3	156	181	0.029	175	0.029	185	0.051	170	0.15	167	0.51
2	1	20	0.6	1	171	227	0.031	220	0.031	229	0.052	193	0.14	183	0.37
2	1	20	0.6	2	167	220	0.032	215	0.035	223	0.052	188	0.11	182	0.42
2	1	20	0.6	3	169	224	0.033	219	0.033	227	0.05	189	0.16	184	0.47
2	1	40	0.3	1	215	268	0.032	270	0.032	279	0.049	233	0.12	231	0.54
2	1	40	0.3	2	219	275	0.031	276	0.031	284	0.048	238	0.17	237	0.62
2	1	40	0.3	3	212	265	0.031	264	0.035	273	0.051	231	0.14	228	0.61
2	1	40	0.6	1	224	278	0.032	267	0.032	286	0.048	263	0.12	260	0.39
2	1	40	0.6	2	227	281	0.033	268	0.033	289	0.051	265	0.19	263	0.64
2	1	40	0.6	3	229	284	0.033	272	0.033	289	0.051	268	0.17	267	0.36
2	3	20	0.3	1	197	256	0.029	247	0.035	263	0.049	218	0.16	203	0.42
2	3	20	0.3	2	202	261	0.031	254	0.031	270	0.051	224	0.15	209	0.52
2	3	20	0.3	3	203	262	0.032	252	0.032	272	0.05	224	0.13	213	0.44
2	3	20	0.6	1	210	250	0.031	247	0.031	262	0.048	232	0.12	230	0.47
2	3	20	0.6	2	206	246	0.032	239	0.032	253	0.048	228	0.16	227	0.42
2	3	20	0.6	3	209	249	0.031	244	0.06	257	0.048	230	0.16	228	0.53
2	3	40	0.3	1	247	301	0.031	296	0.031	318	0.051	270	0.17	265	0.51
2	3	40	0.3	2	249	301	0.032	297	0.032	319	0.051	273	0.17	267	0.5
2	3	40	0.3	3	245	298	0.031	295	0.031	312	0.048	269	0.16	264	0.42
2	3	40	0.6	1	256	318	0.029	309	0.029	332	0.05	283	0.12	280	0.4
2	3	40	0.6	2	258	322	0.03	313	0.033	334	0.048	286	0.13	285	0.38
2	3	40	0.6	3	251	310	0.03	305	0.033	326	0.049	277	0.19	277	0.53

Summary of actual values for N=5, M=2, L=0.3 problems

C	S	P	T	Rep	LB	EDD	TIME	SPT	TIME	LSLK	TIME	BRSP	TIME	SBLIMS	TIME
4	1	20	0.3	1	304	371.184	0.032	365.104	0.031	382	0.049	342	0.16	336	0.39
4	1	20	0.3	2	296	367.928	0.033	358.752	0.035	373	0.051	333	0.17	328	0.64
4	1	20	0.3	3	312	386.568	0.033	381.888	0.033	393	0.05	349	0.17	342	0.36
4	1	20	0.6	1	342	457.254	0.029	448.362	0.032	466	0.048	392	0.16	376	0.42
4	1	20	0.6	2	334	435.536	0.031	439.544	0.031	455	0.048	386	0.12	360	0.52
4	1	20	0.6	3	338	443.794	0.032	447.174	0.035	457	0.048	389	0.13	370	0.44
4	1	40	0.3	1	430	491.49	0.031	500.52	0.032	517	0.051	475	0.19	463	0.47
4	1	40	0.3	2	438	503.262	0.032	511.584	0.033	523	0.051	489	0.16	461	0.42
4	1	40	0.3	3	424	481.664	0.031	484.632	0.033	502	0.048	474	0.15	449	0.53
4	1	40	0.6	1	448	522.368	0.031	512.96	0.035	536	0.05	509	0.13	490	0.51
4	1	40	0.6	2	454	534.812	0.032	526.186	0.031	547	0.048	509	0.12	499	0.5
4	1	40	0.6	3	458	545.936	0.031	532.196	0.032	551	0.049	515	0.16	499	0.42
4	3	20	0.3	1	394	531.506	0.029	523.626	0.031	540	0.044	461	0.14	436	0.4
4	3	20	0.3	2	404	538.128	0.03	546.208	0.032	557	0.048	478	0.11	445	0.38
4	3	20	0.3	3	406	543.634	0.03	546.882	0.06	542	0.051	478	0.16	449	0.53
4	3	20	0.6	1	420	480.06	0.032	463.26	0.031	495	0.052	461	0.12	456	0.41
4	3	20	0.6	2	412	472.976	0.034	462.264	0.032	492	0.052	448	0.17	446	0.37
4	3	20	0.6	3	418	481.536	0.029	471.922	0.031	496	0.05	453	0.14	450	0.51
4	3	40	0.3	1	494	584.896	0.031	583.414	0.029	598	0.049	555	0.12	535	0.37
4	3	40	0.3	2	498	594.612	0.032	595.608	0.033	615	0.048	555	0.19	550	0.42
4	3	40	0.3	3	490	577.71	0.033	587.51	0.033	601	0.051	548	0.17	525	0.47
4	3	40	0.6	1	512	619.008	0.032	598.016	0.032	636	0.048	575	0.14	565	0.54
4	3	40	0.6	2	516	622.296	0.031	609.396	0.034	649	0.051	585	0.15	569	0.62
4	3	40	0.6	3	502	610.934	0.031	593.364	0.029	633	0.051	566	0.15	556	0.61

Summary of actual values for N=5, M=2, L=0.7 problems

C	S	P	T	Rep	LB	EDD	TIME	SPT	TIME	LSLK	TIME	BRSP	TIME	SBLIMS	TIME
2	1	20	0.3	1	164	184	0.031	184	0.035	189	0.051	179	0.19	174	0.64
2	1	20	0.3	2	160	178	0.032	180	0.032	186	0.05	175	0.17	170	0.36
2	1	20	0.3	3	168	185	0.031	188	0.033	194	0.048	185	0.16	180	0.42
2	1	20	0.6	1	183	209	0.032	208	0.033	211	0.048	202	0.15	199	0.52
2	1	20	0.6	2	179	207	0.031	203	0.035	205	0.048	198	0.13	191	0.44
2	1	20	0.6	3	181	209	0.031	205	0.031	209	0.051	201	0.12	194	0.47
2	1	40	0.3	1	227	251	0.032	254	0.032	268	0.051	249	0.16	245	0.42
2	1	40	0.3	2	231	257	0.031	255	0.031	272	0.048	253	0.16	250	0.53
2	1	40	0.3	3	224	251	0.029	248	0.032	263	0.05	247	0.17	242	0.51

2	1	40	0.6	1	236	271	0.03	262	0.06	271	0.048	259	0.17	251	0.5
2	1	40	0.6	2	239	273	0.03	269	0.031	273	0.049	262	0.16	253	0.42
2	1	40	0.6	3	241	274	0.032	273	0.032	278	0.044	265	0.12	257	0.4
2	3	20	0.3	1	209	244	0.034	241	0.031	250	0.048	230	0.13	225	0.38
2	3	20	0.3	2	214	250	0.029	249	0.029	257	0.051	237	0.19	228	0.53
2	3	20	0.3	3	215	250	0.031	248	0.033	254	0.052	237	0.14	232	0.41
2	3	20	0.6	1	222	249	0.032	251	0.033	264	0.052	246	0.15	241	0.37
2	3	20	0.6	2	218	245	0.033	245	0.032	261	0.05	241	0.15	238	0.51
2	3	20	0.6	3	221	249	0.032	247	0.034	259	0.049	243	0.14	239	0.37
2	3	40	0.3	1	259	294	0.031	292	0.029	303	0.048	281	0.11	276	0.42
2	3	40	0.3	2	261	297	0.031	292	0.031	301	0.051	286	0.16	277	0.47
2	3	40	0.3	3	257	291	0.032	286	0.035	299	0.048	282	0.12	274	0.54
2	3	40	0.6	1	268	300	0.033	301	0.033	312	0.051	297	0.17	292	0.62
2	3	40	0.6	2	270	299	0.033	307	0.032	313	0.051	300	0.14	293	0.61
2	3	40	0.6	3	263	293	0.029	293	0.031	304	0.049	292	0.12	285	0.39

Summary of actual values for N=5, M=2, L=0.7 problems

C	S	P	T	Rep	LB	EDD	TIME	SPT	TIME	LSLK	TIME	BRSP	TIME	SBLIMS	TIME
4	1	20	0.3	1	313	351	0.031	351	0.032	361	0.048	342	0.16	331	0.37
4	1	20	0.3	2	305	339	0.032	344	0.031	355	0.05	334	0.14	324	0.51
4	1	20	0.3	3	321	354	0.031	359	0.032	371	0.048	353	0.11	344	0.37
4	1	20	0.6	1	351	401	0.029	399	0.06	404	0.049	387	0.16	381	0.42
4	1	20	0.6	2	343	397	0.03	389	0.031	393	0.044	379	0.12	367	0.47
4	1	20	0.6	3	347	400	0.03	392	0.032	401	0.048	385	0.17	372	0.54
4	1	40	0.3	1	439	486	0.032	491	0.031	518	0.051	482	0.14	475	0.62
4	1	40	0.3	2	447	498	0.034	493	0.029	526	0.052	490	0.12	483	0.61
4	1	40	0.3	3	433	485	0.029	479	0.033	509	0.052	477	0.19	468	0.39
4	1	40	0.6	1	457	524	0.031	507	0.033	525	0.05	501	0.17	486	0.64
4	1	40	0.6	2	463	529	0.032	522	0.032	530	0.049	507	0.14	491	0.36
4	1	40	0.6	3	467	531	0.033	529	0.034	538	0.048	514	0.15	497	0.42
4	3	20	0.3	1	403	470	0.032	465	0.029	482	0.051	444	0.15	435	0.52
4	3	20	0.3	2	413	482	0.031	481	0.031	495	0.048	457	0.16	441	0.44
4	3	20	0.3	3	415	482	0.031	478	0.035	491	0.051	457	0.17	447	0.47
4	3	20	0.6	1	429	481	0.032	485	0.033	509	0.051	475	0.17	466	0.42
4	3	20	0.6	2	421	474	0.033	473	0.032	504	0.049	465	0.16	460	0.53
4	3	20	0.6	3	427	482	0.033	477	0.031	500	0.051	470	0.12	462	0.51
4	3	40	0.3	1	503	571	0.029	567	0.035	588	0.05	546	0.13	535	0.5
4	3	40	0.3	2	507	576	0.031	566	0.032	585	0.048	555	0.19	538	0.42
4	3	40	0.3	3	499	565	0.032	554	0.033	580	0.048	547	0.16	532	0.4
4	3	40	0.6	1	521	582	0.031	586	0.033	606	0.048	577	0.15	568	0.38
4	3	40	0.6	2	525	581	0.032	596	0.035	608	0.051	582	0.13	569	0.53
4	3	40	0.6	3	511	569	0.031	569	0.031	591	0.051	568	0.12	553	0.41

Balas Jobshop Problems

prb	n jobs	n res.	n loops	Optimum	lower bnd		Upper bnd		No. Starts		start soln.	final soln
time	taken	diff	Opt.	improve	Opt	LB	UB					
abz5	10	10	1	1234	nil	nil	20	1316	1251	12.234	17	65	1.38%	
abz6	10	10	1	943	nil	nil	20	1041	943	21.22	0	98	0.00%	
abz7	20	15	1	nil	654	668	20	726	679	39.41	11	47	- 1.68%	1.65%
abz8	20	15	1	nil	635	687	20	729	693	34.221	6	36	- 0.94%	0.87%
abz9	20	15	1	nil	656	707	20	764	715	31.664	8	49	- 1.22%	1.13%

Jaques Carlier Jobshop Problems

car1	11	5	1	7038	nil	nil	20	7092	7041	15.24	3	51	0.04%
car2	13	4	1	7166	nil	nil	20	7194	7169	14.81	3	25	0.04%
car5	10	6	1	7702	nil	nil	20	7741	7709	16.3	7	32	0.09%
car7	7	7	1	6558	nil	nil	20	6603	6560	19.1	2	43	0.03%

S. Lawrence Jobshop Problems

la01	10	5	1	666	nil	nil	20	728	673	18.622	7	55	1.05%
la05	10	5	1	593	nil	nil	20	644	599	12.66	6	45	1.01%
la06	15	5	1	926	nil	nil	20	941	929	14.815	3	12	0.32%
la07	15	5	1	890	nil	nil	20	918	896	11.624	6	22	0.67%
la10	15	5	1	958	nil	nil	20	997	963	21.336	5	34	0.52%
la11	20	5	1	1222	nil	nil	20	1267	1231	17.64	9	36	0.74%
la12	20	5	1	1039	nil	nil	20	1061	1044	14.21	5	17	0.48%

```
la16    10  10  1   945 nil nil 20  1099    961 24.455  16  138 1.69%
la17    10  10  1   784 nil nil 20  856 793 29.313  9   63  1.15%
la20    10  10  1   902 nil nil 20  1134    908 31.372  6   226 0.67%
la21    15  10  1   nil 1040    1053    20  1218    1081    47.322  28  137 - 2.69%   2.66%
la25    15  10  1   977 nil nil 20  1099    991 29.306  14  108 1.43%
la26    20  10  1   1218    nil nil 20  1299    1223    33.269  5   76  0.41%
la27    20  10  1   nil 1235    1269    20  1337    1283    40.355  14  54  - 1.13%   1.10%
la28    20  10  1   1216    nil nil 20  1307    1224    42.697  8   83  0.66%
la29    20  10  1   nil 1120    1195    20  1282    1209    34.23   14  73  - 1.25%   1.17%

Muth and Thompson Jobshop Problems
mt06    6   6   1   55  nil nil 20  59  55  6.334   0   4   0.00%
mt10    10  10  1   930 nil nil 20  943 932 8.42    2   11  0.22%
mt20    20  5   1   1165    nil nil 20  1181    1165    9.446   0   16  0.00%

"Bill Cook, George Steiner Jobshop Problems"
orb1    10  10  1   1059    nil nil 20  1094    1061    7.044   2   33  0.19%
orb2    10  10  1   888 nil nil 20  932 891 6.204   3   41  0.34%
orb5    10  10  1   887 nil nil 20  945 893 7.039   6   52  0.68%
orb8    10  10  1   889 nil nil 20  1014    904 18.924  15  110 1.69%
orb10   10  10  1   944 nil nil 20  1014    952 8.771   8   62  0.85%


Uzsoy Reentrant flow shop problems
problem set re305
prb name    no. mach    no. jobs    no. flex.   no. loops   Lmax known  Algorithm   No. Starts  start soln. final
start final soln  time taken  diff frm. Best  best imp.   improve Best    LB  UB
prob1   5   10  1   4   741 LA2.3   20  883 773 749 18.225  8   24  134 1.08%
prob2   5   10  1   2   552 LA3.5   20  603 556 556 16.31   4   0   47  0.72%
prob3   5   10  1   1   381 ACTS (BF)   20  476 422 391 12.564  10  31  85  2.62%
prob4   5   20  1   4   1092    LA2.3   20  1287    1120    1102    22.475  10  18  185 0.92%
prob5   5   20  1   2   821 LA2.5   20  958 827 827 20.94   6   0   131 0.73%
prob6   5   20  1   1   450 LA3.7   20  596 596 469 17.633  19  127 127 4.22%
prob7   5   10  1   1   270 LA1 20  347 321 274 12.356  4   47  73  1.48%
prob8   5   10  1   2   505 LA3.3   20  612 572 508 15.74   3   64  104 0.59%
prob9   5   10  1   4   1093    LA2.5   20  1146    1183    1097    12.564  4   86  49  0.37%


Chambers Flexible jobshop problems                                                  Gap
Muth and Thompson Jobshop Problems
prb name    no. jobs    no. res.    no. flex.   no. loops   dispatch    Best    Algorithm   No.
Starts  start soln. final soln  time taken  diff frm. Best  improve Best    LB  UB
mt10x   10  10  1   1   1023    922 FRJS    20  971 931 29.662  9   40  0.98%
mt10xx  10  10  2   1   1023    927 FJS 20  1181    943 37.435  16  238 1.73%
mt10xxx 10  10  3   1   1023    918 FJS 20  993 922 51.678  4   71  0.44%
mt10xy  10  10  2   1   982 909 FJS 20  958 919 33.042  10  39  1.10%
mt10xyz 10  10  3   1   937 849 FJS 20  977 883 97.645  34  94  4.00%
mt10c1  10  10  1   1   1023    927 FJS 20  1098    952 34.354  25  146 2.70%
mt10c8  10  10  1   1   1007    913 FJS 20  997 943 37.772  30  54  3.29%
mt10cc  10  10  2   1   980 913 FJS 20  994 937 52.412  24  57  2.63%

S. Lawrence Jobshop Problems (LA24)
setb1x  10  10  1   1   1060    925 FJS/FRJS    20  1022    937 52.412  12  85  1.30%
setb4xx 10  10  2   1   1030    925 FJS 20  1017    958 61.886  33  59  3.57%
setb4xxx    10  10  3   1   1030    925 FJS 20  1044    966 64.325  41  78  4.43%
setb4xy 10  10  2   1   1024    924 FJS 20  1021    969 68.671  45  52  4.87%
setb4xyz    10  10  3   1   1044    914 FJS 20  1009    957 66.258  43  52  4.70%

S. Lawrence Jobshop Problems (LA40)
seti1x  10  15  1   1   1410    1204    FJS/FRJS    20  1389    1274    32.786  70  115 5.81%
seti5xx 10  15  2   1   1393    1204    FJS 20  1377    1281    40.402  77  96  6.40%
seti5xxx    10  15  3   1   1369    1213    FJS 20  1344    1267    64.325  54  77  4.45%
seti5xy 10  15  2   1   1270    1144    FJS 20  1234    1197    31.603  53  37  4.63%
seti5xyz    10  15  3   1   1246    1127    FJS 20  1207    1188    42.741  61  19  5.41%
```