Multiple parallel machines scheduling with setup resources
by Shaowei Wang

A thesis submitted in partial fulfillment Of the requirements for the degree of Master of Science in Industrial and Management Engineering
Montana State University

Abstract:
In this paper a multiple parallel machines scheduling problem with setup resources (MPMSRP) is modeled and studied in which a set of independent tasks need to be processed on a set of renewable resources and nonrenewable resources in a single stage. .Each task can be carried out in several alternative modes; that is, with different resource sets and processing times. The objective is to assign a mode and a start time for each task so that the throughput is maximized. The problem instances are generated and solved by a LP-solver with LP relaxation and a proposed local search heuristic. The computational results for the heuristic are discussed.

MULTIPLE PARALLEL MACHINES SCHEDULING WITH SETUP RESOURCES

by

Shaowei Wang

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 2003

APPROVAL

of a thesis submitted by

Shaowei Wang

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.

Edward L. Mooney, Ph.D. Edward L. Mooney 5/16/03
(Signature) Date

Approved for the Department of Mechanical & Industrial Engineering

Vic A. Cundy, Ph.D. Vic A. Cundy 5-15-03
(Signature) Date

Approved for the College of Graduate Studies

Bruce R. McLeod, Ph.D. Bruce R. McLeod 5-20-03
(Signature) Date

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 to borrowers under 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 permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Signature _____

Date 05/16/03

ACKNOWLEDGEMENTS

I would like to thank Dr. Ed Mooney for his assistance. He gave me this topic and encouraged me to go on thesis study. He did a lot of works in the model development, algorithm and data structure design and coding.

TABLE OF CONTENTS

1. INTRODUCTION 1

2. LITERATURE SURVEY 3

3. NOTATION AND GENERAL PROBLEM FORMULATION 7

4. TEST PROBLEMS 12

5. PRELIMINARY MPMSRP PROBLEM ANALYSIS 16

6. LOCAL SEARCH HEURISTIC 21

7. COMPUTATIONAL RESULTS 28

8. CONCLUSIONS 37

REFERENCES CITED 39

APPENDICES 43

APPENDIX A: GENERATOR AND LOCAL SEARCH SOURCE CODE 44

APPENDIX B: GNU LP-SOLVER SOURCE CODE 84

APPENDIX C: RESULTS OF COMPUTATIONAL EXPERIENCES 96

LIST OF TABLES

Table Page

1. Problem Parameters of the MPMSRP 7
2. Generator Parameter Settings 13
3. Parameters Used to Generate Test Problems 17
4. Lp-Solver Results for T=10, Nr_1=1, Ar_1=1, $d(I)$=[1,10] 18
5. Lp-Solver Results for T=10, Nr_1= Nr_2=1, Ar_1= Ar_2=1, $d(I)$=[1,10] 18
6. Lp-Solver Results for T=10, Nr_1= Nr_2=1, Ar_1=[1,2] Ar_2=1, $d(I)$=[1,10] 19
7. Experiment Problem Sizes 29
8. Results for Gap Analysis 31
9. Results for Computational Experiment 32
10. Results for Instances with n_{m_1} =1 Nr=3 36

LIST OF FIGURES

Figure Page

1. Classical Scheduling Problems Related with MPMSRP 3
2. Example of 3-Way Assignment Process 9
3. The Results of RS_1, RS_2 Two Factors Design. 19
4. A Matrix for a Test Instance (T=10, N=10, Nr_1=Nr_2=1, n_{m_i}=1) 20
5. Search Scheme 22
6. Pseudo Code for Local Search 23
7. Pseudo Code for Multistart 24
8. Moves Neighborhood 25
9. Heuristic Solution Trajectory by Iteration 26
10. Heuristic Solution Trajectory by CPU Time 27
11. Heuristic Gap V.S. Lp Gap 30
12. Impacts of RS 33
13. Impacts of Horizon 34
14. Impacts of Number of Tasks 34
15. Impacts of Running Time 35

ABSTRACT

In this paper a *multiple parallel machines scheduling problem with setup resources* (MPMSRP) is modeled and studied in which a set of independent tasks need to be processed on a set of renewable resources and nonrenewable resources in a single stage. Each task can be carried out in several alternative modes; that is, with different resource sets and processing times. The objective is to assign a mode and a start time for each task so that the throughput is maximized. The problem instances are generated and solved by a LP-solver with LP relaxation and a proposed local search heuristic. The computational results for the heuristic are discussed.

CHAPTER 1

INTRODUCTION

Scheduling concerns allocating limited resources to tasks over time. In this paper, we consider a multiple parallel machines scheduling problem in which three types of resources are available over time: *renewable parallel resources* (R_1), *renewable setup resources* (R_2) and *nonrenewable (consumable) setup resources* (R_3). Renewable (parallel and setup) resources are assumed available at every time period. Examples would be manpower and tools. Railcars may be modeled as parallel, renewable resources and setup tools and operators are examples of renewable setup resources.

In contrast to the renewable resources, nonrenewable resources are consumable over time periods with a limited total consumption. An example would be raw materials. The setup resources are consumed in 1 period.

The study was motivated by a railcar scheduling problem studied by Li [1]. In this model, the railcars were renewable parallel resources, the loading facilities were renewable, setup resources and the inventories of material were the consumable resources.

A set of tasks has to be carried out without preemption using a specified set of resources, or *mode*. Each task can be performed in one out of a set of alternative modes with a processing time specified for each mode. We assume that there is no precedence

constraints between tasks and the setup resources are requested for only one time period. The goal is to choose a mode and a start time for each task so that the throughput is maximized over a given planning horizon.

This paper defines a model for the *Multiple Parallel Machines with Setup Resources Problem* (*MPMSRP*) and explores the relationship of the problem instance characteristics to the solution methods. The problem instances are generated and solved by a LP-solver with LP relaxation and a proposed local search heuristic. The computational results are discussed.

The paper is organized as follows. In Chapter 2, we introduce several classical scheduling problems that are related to MPMSRP and address the differences. In Chapter 3, we present the notation and formulate three models. In Chapter 4, a test problem generator is given. In Chapter 5, we discuss some properties of the model and a local search heuristic is proposed in Chapter 6. Then the experimental design and the performance of proposed heuristic are covered in Chapter 7. Finally, Chapter 8 provides a brief summary and conclusions.

CHAPTER 2

LITERATURE SURVEY

The multiple parallel machines scheduling with setup resources problem (MPMSRP) is a restriction of the general resource-constrained scheduling problem without precedence constraints and a generalization of several other problems. There are four types of classical scheduling problems that are most closely related with the MPMSRP: Parallel Machines, Flexible Job/Flow Shop Scheduling, Routing & Scheduling (Transportation, Logistics) and Resource Constrained Project Scheduling (RCPSP). See Figure 1 for detail.

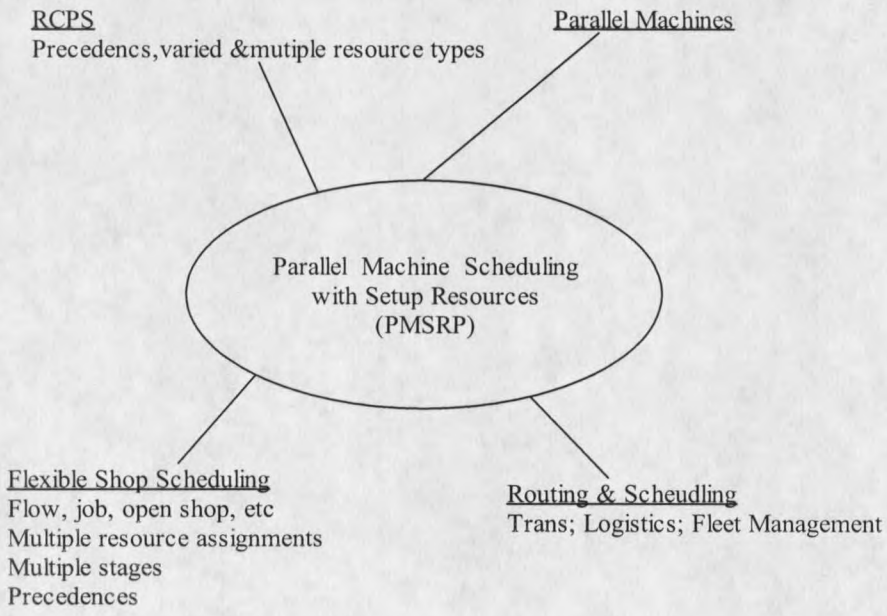

Figure 1. Classical Scheduling Problems Related with MPMSRP

In the classical parallel machine scheduling problem only parallel (renewable) machine capacity is considered; no additional resource types are considered. Daniels et al.[2] explore the impact of resource flexibility by developing and analyzing heuristics for the identical parallel-machine flexible-resource scheduling problem with unspecified job assignment (UPMFRS). No setup resources are considered here, a mixed-integer program is used for model formulation.

A few scheduling models have appeared in the literature which take additional resources into consideration. Ventura and Kim[3] identified a problem of scheduling jobs on parallel machines with an unrestricted due date and additional resources. They considered only a single type of additional resource and each job requires one machine and at most one unit of additional resource for it's processing. Bourland et al. [4] discussed the fractional setup operator requirements in processing a task. In their studies, the operator can handle many machines. They grouped the machines in different stages in order to find the optimum number of operators required to process all tasks. Since the workers movement is limited to one stage, grouping is confined to single stage. They concluded that this approach might not be feasible for realistically sized problems. There is a finite planning horizon of t periods. Setup times are assumed to be integer multiples of one period. The operator is treated as a setup resource type. Slowinski [5] considered a parallel machine scheduling problem where the jobs may require an additional renewable resource, but he assumed setup time is zero and allows pre-emption without penalty.

Herrmann and Lee [6] examined parallel machine scheduling with operator constrained setups. Only one setup resource is considered. Olafsson and Shi[7] address the Parallel-Machine Flexible-Resource Scheduling (PMFRS) problems of simultaneously allocating flexible resources and sequencing jobs. One renewable resource is considered.

Flow shop, job shop and flexible shop scheduling problems are generalizations of our problem where jobs require more than one group of parallel machines. In flow shop problems, all the jobs follow the same sequence. In Job shop problems, all the jobs can have different sequences. Both require multiple resources but with only one request unit. In flexible flow shop problem, like Yang, S.K.a.M.P. [8] Chung and Shi [9], one or more resource units are required. Daniels and Mazzola [10] consider a scheduling problem in which the tasks follow the same sequence. The best sequence that minimizes the schedule makespan is found using the iterative procedure. Only one resource is studied here .The resource is allocated by assigning an integer number of workers to a job. They concluded that the complexity reduces the optimality of large practical problems. Our problem differs from Daniels and Mazzola's problem as each task in our problem requires multiple resources. Brah and Loo [11] studied the flow shop with multiple processors problem. It is a generalization of flow shop problem, where at least one stage the processor has more than one identical machine. The jobs are subject to precedence constraints. No setup resource constraint is considered.

Routing and Scheduling problems (Transportation, Logistics, etc) are related to our problem. Sherali et al. [12] mainly focused on the Kuwait Petroleum Corporation(KPC) Problem. Crude oil and a number of refined oil-related products are shipped to customer. Two classes of vessels are considered: the first is the fleet of vessels controlled by KPC, the second is spot-chartered. A mixed-integer programming model is constructed. No setup resource is considered. Powell et al. [13] develop a new method for solving Dynamic Resource Allocation Problems(DRAP). Xu and Kelly [14] use network flow-based tabu search to solve the problem. Such problems focus on the allocation of resources to perform tasks over a network, for example, routing and scheduling over 5,000 drivers to serve 30,000 loads over a four days horizon.

In the literature, scheduling problems where multiple resources are considered are known as resource-constrained project scheduling problems (RCPSP) and have received considerable attention during the past several decades. Tasks are subject to precedence relations, require units of multiple renewable, non-renewable constrained resources and can be performed in multiple modes. Like Salewski et al. [15], Shewchuk and Chang [16], Bert and Eilly [17] and Rainer and Andreas [18], etc. In our case, all the jobs have no precedence.

CHAPTER 3

NOTATION AND GENERAL PROBLEM FORMULATION

The notation used to model the parallel machine scheduling with setup resources (MPMSRP) is summarized in Table 1.

Table 1. Problem Parameters of the MPMSRP

Problem Notation	Definition
$T=\{t: t=1,2,\ldots n_t\}$	Set of time period(planning horizon) indexes
$I=\{i: i=1,2,\ldots n_i\}$	Set of task indexes
$M_i=\{m: =1,2,\ldots n_{m_i}\}$	Set of mode indexes for task i
$d(i,m)$	Duration of the i^{th} task in mode m
$R=\{r: r=1,2,\ldots n_r\}$	Set of resource indexes $r\in R=\{R_1\|R_2\|R_3\}$
R_1	parallel, renewable resources with resource number $NR_1=\|R_1\|$
R_2	setup, renewable resources with resource number $NR_2=\|R_2\|$
R_3	setup, consumable resources with resource number $NR_3=\|R_3\|$
A_r	Resource capacity for resource r
a_{imr}	Number of units of resource r required in mode m for task i

We define a model for the problem of simultaneously sequencing tasks on multistage parallel, identical resources and scheduling setup resources over a planning horizon consisting of n_t time periods

Sets of discrete renewable and consumable resources are available. Each renewable resource $r\in R_1 \cup R_2$ is always available. Each consumable resource $r\in R_3$ is only available over the whole horizon and is consumed over time. Each task i can be carried out in

several modes, and each mode requires a set of resources $R_m \subseteq R$ with a mode-dependent processing time. Here we assume $d(i,m)$ is the duration of task i performed in mode m. During this time, the renewable parallel resources R_1 are used, while for renewable (R_2) and consumable setup resources (R_3) only one time period is assumed. Each mode is treated as different from any other mode, even if two modes of different tasks are assigned with the same resource set and with the same duration.

Many linear objective functions can be selected, such as minimizing makespan, minimizing total tardiness, etc. In many manufacturing systems, system throughput is very important. In this paper, the objective is to schedule each task in one of its modes, subject to the resource constraints under the objective of maximizing the throughput during a fixed horizon.

Three different MPMSRP models are given below ($MPMSRP_1$, $MPMSRP_2$, and $MPMSRP_3$). $MPMSRP_1$ is a 3-way assignment model. It uses 0-1 variables to formulate the model using the general idea given in Pritsker, Watters, and Wolfe [19]. Using this model, the process of finding an optimal solution is represented as a 3-way assignment: a mode and a start time are assigned to tasks. In mode assignment, a specific resource set (mode), is assigned to each task. After that tasks with fixed mode are assigned to different start time to maximize the throughput. An example is showed in Figure 2.

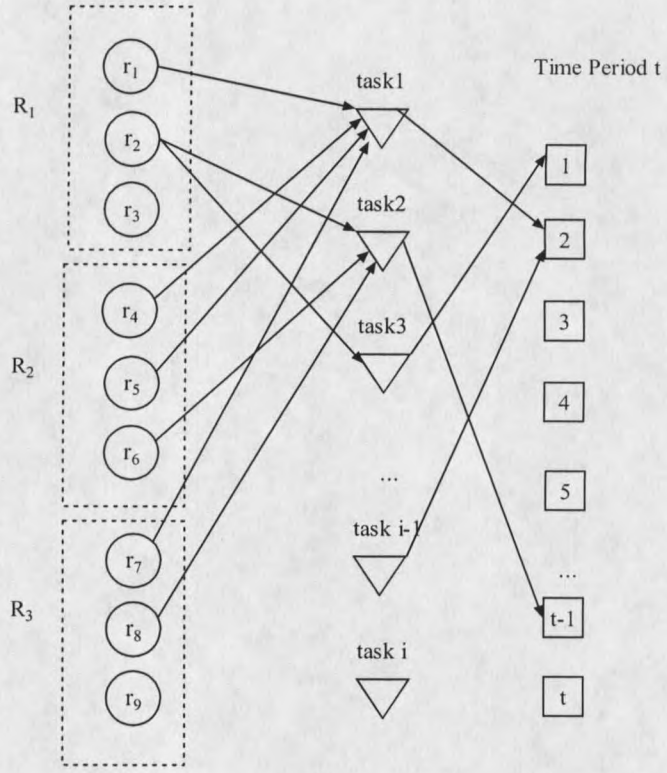

Figure 2. Example of 3-way assignment process

We wish to choose "optimal" values of decision variables x_{imt}, where $m \in M_i$, and

$$x_{imt} = \begin{cases} 1 & \text{if task } i \text{ is assigned to mode } m \text{ at start of period } t \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

The problem can be formulated as follows:

$$Max \sum_{i=1}^{I} \sum_{m=1}^{N_{mj}} \sum_{q=1}^{T} x_{imq} \quad (2)$$

Subject to:

$$\sum_{m=1}^{n_{mj}} \sum_{q=1}^{t} x_{imq} \leq 1 \qquad \text{for all } i \in I \quad (3)$$

$$(\text{MPMSRP}_1)\ \sum_i \sum_{\substack{m\in M_{ir},and \\ t-d(im)<q\le t}} a_{imr}x_{imq} \le A_r \quad \text{for all } r\in R_1, t\in T \tag{4}$$

$$\sum_i \sum_{m\in M_{ir}} a_{imr}x_{imt} \le A_r \quad \text{for all } r\in R_2, t\in T \tag{5}$$

$$\sum_i \sum_{m\in M_{ir}} \sum_t a_{imr}x_{imt} \le A_r \quad \text{for all } r\in R_3 \tag{6}$$

The objective function (2) maximizes the throughput. Constraint (3) ensures that at most one mode is assigned to any task. This also makes sure that each task can be assigned at most once. Constraint (4) guarantees that parallel, renewable resources assigned at each time period do not exceed their capacity. Constraint (4) also ensures that each task must be continuously processed to its completion without preemption. We assume setup, renewable resource and setup consumable resources only need one time period for processing. So constraint (5) ensures that at any time period the assignment of setup, renewable resources cannot exceed its capacity. Constraint (6) ensures that the total consumption of setup, consumable resources cannot exceed its total capacity for the whole horizon.

We also define a two-way assignment model. MPMSRP_2 represents the problem as the assignment of slots to task where slots represent feasible start time-mode pairings.

$$x_{is} = \begin{cases} 1 & \text{if task } i \text{ is assigned to slot } s(m,t) \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

$$Max \sum_{i=1}^{I} \sum_{s=1}^{S_i} x_{is} \tag{8}$$

Subject to:

$$\sum_{s=1}^{S_i} x_{is} \le 1 \text{ for all } i \in I \tag{9}$$

(PSMRP$_2$)
$$\sum_{i} \sum_{\substack{s \in S_{ir}, \\ \text{and } t-d(im(s))<q\le t}} a_{im(s)r} x_{is} \le A_r \quad \text{for all } r \in R_1, t \in T \tag{10}$$

$$\sum_{i} \sum_{s \in S_{ir}} a_{im(s)r} x_{is} \le A_r \quad \text{for all } r \in R_2, t \in T \tag{11}$$

$$\sum_{i} \sum_{s \in S_{ir}} \sum_{t} a_{im(s)r} x_{is} \le A_r \quad \text{for all } r \in R_3 \tag{12}$$

We also proposed a model MPMSRP$_3$ as follows. Unlike the MPMSRP$_1$, by using this model, we can easily code the heuristics and simplify the search base on simple complement moves.

$$x_j = \begin{cases} 1 & \text{if task } i(j) \text{ is assigned to choice } j \\ 0 & \text{otherwise} \end{cases} \tag{13}$$

where each j indexes a choice triple (i (j), m (j), t (j))

$$Max \sum_{j=1}^{J} x_j \tag{14}$$

Subject to:

$$\sum_{j(i(j))} x_j \le 1 \quad \text{for all } i \in I \tag{15}$$

(MPMSRP$_3$)
$$\sum_{j \in J_r} \sum_{t-d(i(j)m(j))<q\le t} a_{r(m(j))} x_j \le A_r \quad \text{for all } r \in R_1, t \in T \tag{16}$$

$$\sum_{j \in J_r} a_{r(m(j))} x_j \le A_r \quad \text{for all } r \in R_2, t \in T \tag{17}$$

$$\sum_{j \in J_r} \sum_{t} a_{r(m(j))} x_j \le A_r \quad \text{for all } r \in R_3 \tag{18}$$

CHAPTER 4

TEST PROBLEMS

We need some test instances to investigate the properties of our proposed models and testing the proposed algorithm. Normally, two possible approaches can be used to come up with test instances: First, we could use data from real-world cases. However, even if an algorithm performs well on some instances, it does not guarantee that it will perform well on other instances. A second approach is to generate artificial instances. If a tough test instance can be solved well by an algorithm, it is likely that a real-world instance could be solved as well.

Therefore, we decided to test our algorithm with randomly generated test instances and a set of factors were proposed as the parameters to control the experiments. Here we assume that an instance is determined by the length of the planning horizon, the number of tasks, the number of modes, *resource strength* (*RS*) and *resource factor* (*RF*).

All the values and parameters used in our experiment are described in Table 2. RS_k, the resource strength of resources of type k=1, 2, or 3, is defined by Kolisch et al.(1996)[20].

$$RS_k = (A_r - a_r^{\min})/(a_r^{\max} - a_r^{\min}) \qquad r \in R,\ k = 1,2,3 \tag{19}$$

RS_k is used to determine resource capacity, A_r, for each resource.

$$A_r = a_r^{\min} + RS_k \times (a_r^{\max} - a_r^{\min}) \qquad r \in R,\ k = 1,2,3 \tag{20}$$

Table 2. Generator parameter settings

Parameter	Value
T (Horizon)	10;20
n_i (Number of tasks)	10;20;30;40;50
n_{m_i} (Number of modes per task)	1;2;3
$d(i,m)$ Duration of the ith task in mode m	[1,10]
r Number of parallel, renewable resource (R_1)	[3,3]
$\neq$ Number of Setup, renewable resource (R_2)	[3,3]
$\neq$ Number of Setup, consumable resource (R_3)	[3,3]
a_{imr} (parallel, renewable resource demand)	[1,10]
$a_{im\neq}$ (setup, renewable resource demand)	[1,10]
$a_{im\neq}$ (Setup, consumable resource demand)	[1,10]
RF_1 (resource factor for R_1 type)	1
RS_1 (resource strength for R_1 type)	0.1;0.25
RF_2 (resource factor for R_2 type)	1
RS_2 (resource strength for R_2 type)	0.1;0.25
RF_3 (resource factor for R_3 type)	1
RS_3 (resource strength for R_3 type)	0.1;0.25

Note: the indication $[x,y]$ means uniform distribution between x,y

Setting RS_k=0 will let $A_r = a_r^{\min}$ and give the smallest feasible resources for that resource type $r \in R$, whereas, setting RS_k=1 gives the largest feasible resources of $a_r^{\max}$ with no resource constraint. Experience showed that, RS_k>0.5 makes test instances very easy, so here we only choose RS_k=0.1 and 0.25.

For the renewable resource type $r \in R_1 \cup R_2$, the lowest availability $a_r^{\min}$ is obtained by setting all the tasks a mode with the lowest demand and selecting the largest one, that is,

$$a_r^{\min} = \max_{i=1}^{n_i} \min_{m=1}^{n_{m_i}} \{a_{imr}\} \quad , r \in R_1, R_2 \tag{21}$$

The maximum level of $a_r^{\max}$ is the peak demand for renewable resource type $r \in R_1 \cup R_2$. Since there is no precedence between tasks, $a_r^{\max}$ is obtained by assigning all the tasks at the same time in a mode with the largest per-period demand for the resource type r and calculate the peak demand, that is,

$$a_r^{\max} = \sum_{i=1}^{n_i} \max_{m=1}^{n_{m_i}} \{a_{imr}\} \quad , r \in R_1 \cup R_2 \tag{22}$$

For a nonrenewable setup resource r, $r \in R_3$,

$$a_r^{\max} = \sum_{i=1}^{n_i} \max_{m=1}^{n_{m_i}} \{a_{imr}\} \quad , r \in R_3 \tag{23}$$

$$a_r^{\min} = \sum_{i=1}^{n_i} \min_{m=1}^{n_{m_i}} \{a_{imr}\} \quad , r \in R_3 \tag{24}$$

The resource factor, *RF*, (Pascoe,1966[21]) reflects the average portion of resources used or consumed. *RF*=1 means each task demands every resource, whereas *RF*=0 indicates a problem without resource demands. Equation (25) defines RF_k for resource type k=1, 2, 3.

$$RF_k = \frac{1}{n_i} \sum_{i=1}^{n_i} \frac{1}{M_i} \frac{1}{|R|} \sum_{m=1}^{n_{m_i}} \sum_{r=1}^{|R|} \begin{cases} 1 & if \quad a_{imr} > 0, \\ 0 & otherwise. \end{cases} \tag{25}$$

For our experiments, test instances were generated using the following procedure with the parameters above.

First, given the upper and lower levels shown in Table 2, the number of resources for each resource type was randomly chosen from Uniform [Lower, Upper]. With the number of tasks n_i and upper and lower level of the number of mode, for each task, the number of modes was randomly selected from Uniform [Lower, Upper]. For each mode, its duration was randomly chosen from uniform [Lower, Upper]. Then using the resource factor, RF_k, for each resource type and equation (25), a resource usage matrix was generated and resource units were randomly assigned from uniform distribution between the lower and upper. Finally, given the resource strength RS for each resource type, the capacity of each resource was calculated by equations (20) through (24).

CHAPTER 5

PRELIMINARY MPMSRP PROBLEM ANALYSIS

The 0-1 ILP formulation in Chapter 3 precisely models the problem and is equivalent to the problem:

$$\max\{cx \mid Ax \le b, x_j = 0 \text{ or } 1, j \in N\}, \tag{26}$$

Where A is $m \times n$, b is $m \times 1$, and $N=\{1,\ldots n\}, M=\{1,\ldots,m\}$. The LP relaxation of this problem can be solved with the simplex method. We found that the optimal relaxation solution is often integer when a_{imr} is 0 or 1 and A_r is integer. We began by exploring the relationship between resource constraints and integer feasible LP relaxations.

We already knew the problem was NP-hard. Bianco et al. [22] proved that even in the simplest case, if a single mode is given for each task, the multiple modes scheduling problem (MMSP) without precedence is NP-hard. MMSP is a restriction of our problem, so MPMSRP is NP-hard. No efficient solution with an exact algorithm would likely give optimal solutions in a reasonable time. We confirm these properties with experiments involving LP relaxation. Since no benchmark problems are available, we use randomly generated problem instances to test the integer property.

As discussed in Chapter 4, the resource capacity (A_r) relative to demand (a_{ir}) is controlled by *RS*. We chose two levels for *RS*, loose capacity and tight capacity, corresponding to *RS*=0.5 and 0.1 respectively for this study.

To illustrate, consider the simplest case: single mode (n_{m_i}=1), one type of parallel, renewable resource ($r_1 \in R_1$) and one type of renewable, setup resource ($r_2 \in R_2$). Table 3 shows the parameters used to generate test problems. Different parameter combinations are used to randomly generate 5 test instances.

Table 3. Parameters used to generate test problems

Problem Parameter	Values considered	Total
Planning horizon ($\|T\|$)	10	1
Number of tasks ($\| n_i \|$)	10,20,30	3
Duration of task i, $d(i)$	uniform[1-10]	

To test the integer property, here we use an *Integrality Index* obtained by solving the LP relaxation with GLPK (GNU Linear Programming Kit) 3.23.

$$Integrality\ Index = \frac{Total\ Number\ of\ Integer\ results}{Total\ Number\ of\ Variables} \times 100$$

Table 4 shows results for one parallel renewable resource with NR_1=1, T=10, ar_1=1, $d(i)$=[1,10] and different number of tasks (problem size). The first column is the number of tasks. The second column is the resource strength RS and total resource capacity. There are five replications for each case. In some cases, we get integer LP relaxation. In general, the tighter the constraints are, the lower the percentage of integer results is.

Table 4. LP-solver Results for T=10, NR_1=1, ar_1=1, $d(i)$=[1,10]

n_i	RS/A_{r1}	Cons.	Var.	Integrality Index(Rep.)					LP relaxation value(Rep.)				
				1	2	3	4	5	1	2	3	4	5
10	0.5/6	20	100	100	92	94	100	100	10	10	10	10	10
	0.1/2	20	100	86	89	88	86	91	7.5	8.8	8.3	6	7
20	0.5/11	30	200	100	100	100	100	100	10	10	10	10	10
	0.1/3	30	200	94	90	92	97	93	14	11	12.7	11	11
30	0.5/16	40	300	97	97	96	97	97	30	30	30	30	30
	0.1/4	40	300	92	96	93	95	97	20.4	15	16.6	18.8	19

It is not always the case that a tighter constraint gives a lower percentage of integer results. As seen from the results in Table 5, instances with one parallel, renewable resource and one additional renewable setup resource (NR_1= NR_2=1) and one required unit for each type of resource (ar_1= ar_2=1), as setup resource constraints become tighter, the relaxation solution becomes "more integral." If we increase the required units for each type of resource to uniform [1, 2], we get the same results, as shown in Table 6.

Table 5. LP-solver Results for T=10, NR_1= NR_2=1, ar_1= ar_2=1, $d(i)$=[1,10]

n_i	RS_1 /A_{r1}	RS_2 /A_{r2}	Const.	Var	Integrality Index (Rep.)					LP relaxation value (Rep.)				
					1	2	3	4	5	1	2	3	4	5
10	0.5/6	0.5/6	30	100	100	100	94	92	100	10	10	10	10	10
	0.1/2	0.1/2	30	100	89	89	89	82	91	7.5	8.8	8.3	6	7
	0.5/6	0.1/2	30	100	100	100	100	89	100	10	10	10	10	10
20	0.5/11	0.5/11	40	200	100	100	100	100	100	20	20	20	20	20
	0.1/3	0.1/3	40	200	94	90	92	97	93	14	11	12.7	11	11
	0.5/11	0.1/3	40	200	100	100	100	100	100	20	20	20	20	20
30	0.5/16	0.5/16	50	300	97	97	96	97	98	30	30	30	30	30
	0.1/4	0.1/4	50	300	92	96	93	95	97	20.4	15	16.6	18.8	19
	0.5/16	0.1/4	50	300	98	100	96	100	100	30	30	30	30	30

Using data from Table 5 and Table 6 and treating RS_1, RS_2 as factors, we analyzed the impact of RS_1, RS_2 by two level factorial experiments. The ANOVA showed that with RS_1 fixed at 0.5 and RS_2 decreased from 0.5 to 0.1, the increase of average percentage of integer results is significant (α=5%). Figure 3 shows this result graphically. We can also confirm the general MPMSRP lack the integrality property.

Table 6. LP-solver Results for T=10, NR_1= NR_2=1, ar_1=[1,2] ar_2=1, $d(i)$=[1,10]

n_i	RS_1 /A_{r1}	RS_2 /A_{r2}	Const.	Var	Integrality Index (Rep.)					LP relaxation value (Rep.)				
					1	2	3	4	5	1	2	3	4	5
10	0.5/7]	0.5/6	30	100	91	89	89	88	87	10	10	10	10	10
	0.1/4	0.1/2	30	100	83	81	82	86	84	10	9.9	8.9	7.3	9.5
	0.5/6	0.1/2	30	100	96	96	98	94	92	10	10	10	10	10
20	0.5/16	0.5/11	40	200	94	93	93	93	96	20	20	20	20	20
	0.1/5	0.1/3	40	200	83	89	89	89	89	16	12.8	14.8	13.9	14.4
	0.5/16	0.1/3	400	200	97	98	97	96	98	20	20	20	20	20
30	0.5/26	0.5/16	50	300	96	94	96	96	96	30	30	30	30	30
	0.1/7	0.1/4	50	300	94	93	93	93	97	21.9	18.8	20.1	20.7	23.5
	0.5/26	0.1/4	50	300	96	94	98	98	99	30	30	30	30	30

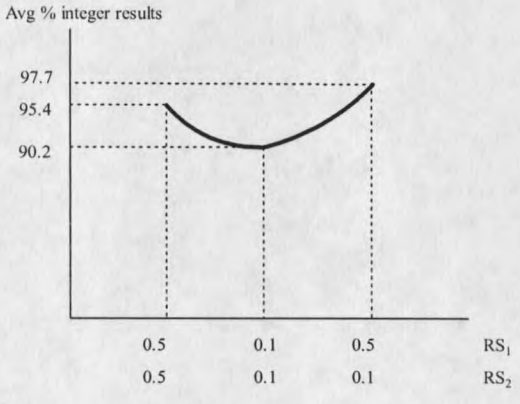

Figure 3. The results of RS_1, RS_2 two factors design.

However, as setup resource constraints become tighter, the solution of the relaxation becomes "more integral". This is true for single and multi-mode problems. If all the setup resources (R_2) constraints become tighter than parallel, renewable (R_1), it becomes a transportation problem. The LP relaxation for a transportation problem always yields integer results. Figure 4 shows the A matrix for a MPMSRP test instance with one parallel, renewable resource and one setup, renewable resource, T=10, N=10, n_{m_i}=1. If the parallel, renewable, resource constraints are removed, the transportation structure is apparent.

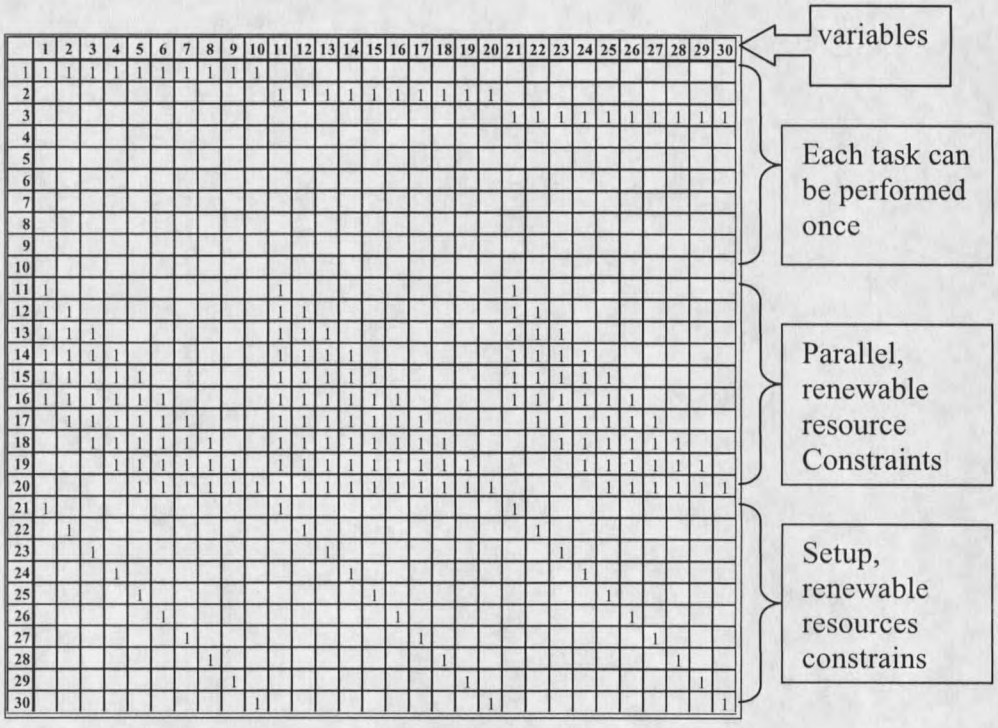

Figure 4. A matrix for a test instance (T=10, N=10, NR_1=NR_2=1, n_{m_i}=1)

CHAPTER 6

LOCAL SEARCH HEURISTIC

LP solvers have been used to solve small, special cases of test problems. We notice that for a small size problem, it is easy to get an optimum solution using a general implicit-enumeration procedure. However, it takes a prohibitively long time to get the optimum solution for large-scale problems as expected from preliminary study.

For example, given tasks of N=50 and the planning horizon T=10, for each task with n_{m_i}=3 possible modes, the total number of possible solution is:

$$M \times T \sum_{i=1}^{N} C_N^i = 3 \times 10 \times 2^{50} = 33{,}776{,}997{,}205{,}278{,}700 \tag{27}$$

Using the GNU MIP solver, after 10 hours running on a 1.66 GHz CPU, no optimal solution was found. As we mentioned in Chapter 5, the MPMSRP problem is NP-hard, the use of heuristic procedures should be investigated. Here we present a local search heuristic to get a local optimum solution and a multistart descent method to get the global optimum or near optimum solution.

Figure 5 shows the general multistart scheme for a discrete search space. After a series of improving moves, local optimum solution for space S1 can be obtained and saved as the best solution ever obtained so far. Starting from another initial solution and performing local search for space S2, we can get another local optimum. If this solution

is better than the best solution so far, we save it as the best solution. Continue searching until no further improvement or the specified number of cycles reached.

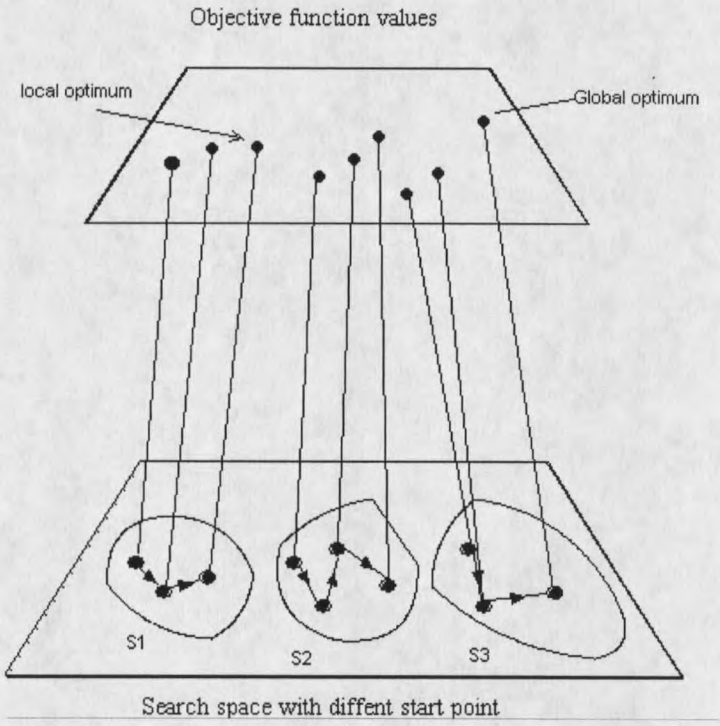

Figure 5. Search Scheme

Here we use model $PMRSP_3$ in chapter 3 to represent the solutions and to form a search procedure. We introduce a concept of "Choices" x_j, an (i, m, t) triple with 0-1 value. Each task has a choice list, which includes all the possible choices for that task.

Four types of moves that form the neighborhood for the heuristic were developed in this paper:

1. Insert choices of unscheduled tasks
2. Remove scheduled choices
3. Swap unscheduled and scheduled choices
4. Move scheduled choices

The pseudo code in Figure 6 illustrates the basic improvement algorithm. By starting from a randomly generated initial solution, a sequence of solutions was obtained by repeatedly moving from a current solution to a solution in an appropriately defined neighborhood. In this paper, the deepest descent method was used to evaluate all the moves in the neighborhood and select and implement the one with the best improving move. The resulting solution is a local optimum.

```
Random initial solution;
Local_improve=objective function;
While (Local_improve>0)
{
    moves;
    evaluate moves
    ....
    Calculate Local_Improve;
```

Figure 6. Pseudo code for Local Search

Even though the local search ensures that a local optimum for an initial solution can be obtained, it cannot guarantee that a global optimum can be reached. Multistart search overcomes this problem by conducting local search from more than one random initial

starting solutions and returning the best local optimum. The pseudo code in Figure 7 shows basic multistart component.

```
Loop for a desire number of cycles
{
    Random create a new initial solution;
    Get the local optimum solution;
    Compare and save the best solution;
}
```

Figure 7. Pseudo code for Multistart

To evaluate the moves, we rewrite and add the constraint violations to the objective function with a negative penalty as follows:

Maximize Z= Number of Assignments – (capacity violations)* Penalty (28)

$$Capacity\ Violations = \sum_{t=1}^{T} \sum_{r \in R} Max\left\{0, \sum_{i} \sum_{m \in M_{ir}} a_{imr} x_{imt} - A_r\right\}. \quad (29)$$

A solution is feasible only if capacity violations equals zero.

Lexicographic rules were used to implement the four types of moves for different purposes. Our search normally oscillates between insert moves and swap moves. The Insert moves are executed to increase the number of assignments, while swap moves are performed to balance the resource loading and, hopefully, open up opportunities for more inserts.

As shown in Figure 8, the search begins with insert moves. First we want to increase the Z value by trying to insert choices for unscheduled tasks. If we cannot increase the Z value with inserts, we will try to swap unscheduled choices with all scheduled choices making feasible swaps that will average out the resource usage. By removing a scheduled choice with high resource usage, the chance to insert an unscheduled choice should increase. Finally, if there are no unassigned tasks or all unassigned choices have been evaluated, then we have a local optimum and the procedure stops. Otherwise the cycle repeats.

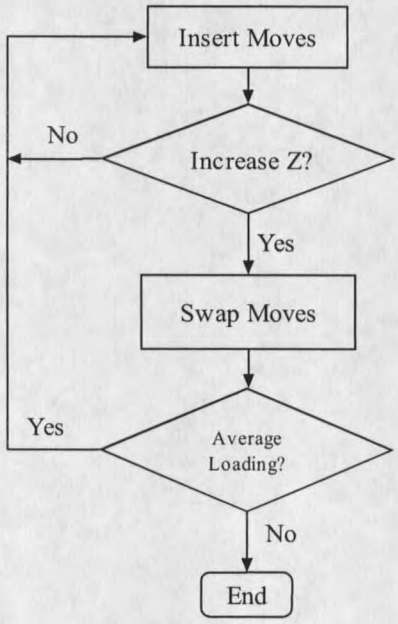

Figure 8. Moves neighborhood

The load for resource r, L_r, can be computed using equations (30) to (32).

$$L_r = (\sum_i \sum_{\substack{m \in M_{ir}, and \\ 1 \le q \le T}} d(im) \times a_{imr} x_{imq}) / (T \times A_r) \qquad \text{for all } r \in R_1 \tag{30}$$

$$L_r = (\sum_i \sum_{m \in M_{ir}, 1 \le q \le T} a_{imr} x_{imq}) / (T \times A_r) \qquad \text{for all } r \in R_2 \tag{31}$$

$$L_r = (\sum_i \sum_{m \in M_{ir}, 1 \le q \le T} a_{imr} x_{imq}) / (T \times A_r) \qquad \text{for all } r \in R_3 \tag{32}$$

Since the swap neighborhood is very large, it is time-consuming to evaluate all the possible swaps. Therefore, we evaluate swaps by first ordering resources in decreasing L_r. We check tasks assigned to resources until a feasible swap is found or determine that no feasible swaps are possible.

Figure 9 is an example of the heuristic solution trajectory by iteration for an instance with n_i=50, n_{m_i}=1, T=10, RS=0.1. The LP bound is 28.84. The initial solution is 12. After 1839 insert moves and 45360 swap moves, it reaches a local optimum of 21.

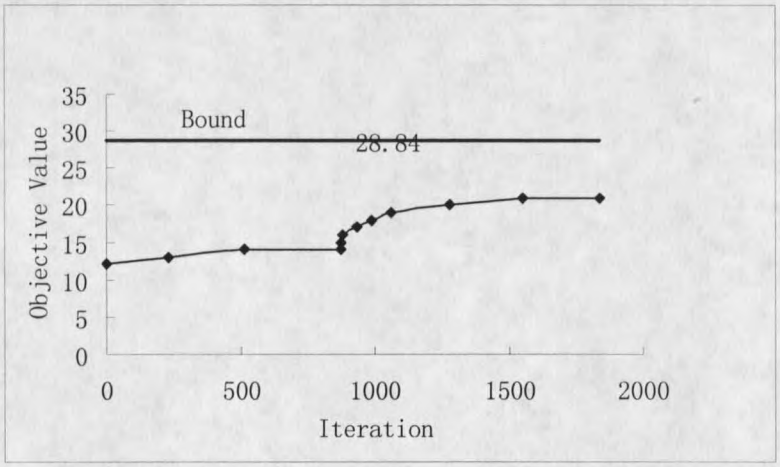

Figure 9. Heuristic solution trajectory by iteration

Figure 10 shows an example of the heuristic solution trajectory by CPU time for an instance with n_i=50, n_{m_i} =1, T=10, RS=0.1. The LP bound is 28.84. The initial solution is 12. After running 0.06 second insert moves, it changes to swap moves and takes 13.58 seconds of swap moves to average the resource usage and 0.06 second of insert moves to reach a local optimum of 21.

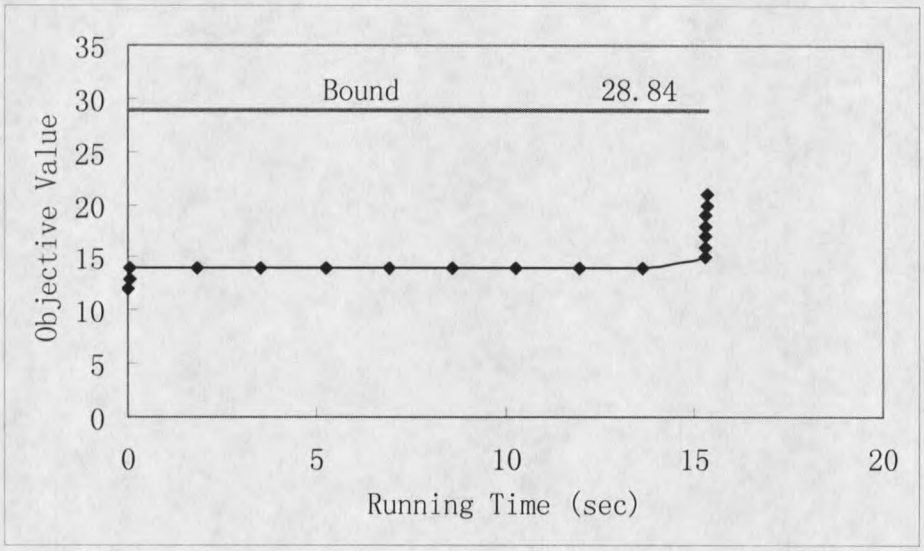

Figure 10. Heuristic solution trajectory by CPU time

The insert move evaluation has a computational complexity of $O(n_i)$ per iteration, while the swap moves have an $O(n_i^2)$ complexity. Since most of the time is spent evaluating moves, the time complexity is between $O(n_i)$ and $O(n_i^2)$ for our heuristic.

CHAPTER 7

COMPUTATIONAL RESULTS

This chapter documents the performance of the proposed heuristic over a range of problem instances. The results were obtained using 1.66GHz Athelon (AMD) running REDHAT Linux 7.2. The algorithms and test instances generator were coded in GNU GCC2.96-98, GLIB 1.2.10 and Standard Template Library (STL). The GNU Linear Programming Kit (GLPK), Version 3.2.3, was used as an LP-solver.

As shown in Table 7, testing was performed for problem instances of different sizes with varying number of modes. Each problem size was run for two resource strength levels, resulting in 60 cases. Tests were conducted with 3 of each resource type, two levels of RS (0.1 and 0.25) and RF=1. There are 5 replicates for each run, for a total of 300 instances. The stopping criterion was set to 30 seconds running time for all instances. The total number of variables and constraints for each instance varied with the horizon, task number and mode number. Detailed computational results for some instances can be found in Appendix C.

Table 7. Experiment Problem Sizes

Horizon	Task Number	Modes Number	Variables	Constraints
10	10	1	100	73
		2	200	73
		3	300	73
	20	1	200	83
		2	400	83
		3	600	83
	30	1	300	93
		2	600	93
		3	900	93
	40	1	400	103
		2	800	103
		3	1200	103
	50	1	500	113
		2	1000	113
		3	1500	113
20	10	1	200	133
		2	400	133
		3	600	133
	20	1	400	143
		2	800	143
		3	1200	143
	30	1	600	153
		2	1200	153
		3	1800	153
	40	1	800	163
		2	1600	163
		3	2400	163
	50	1	1000	173
		2	2000	173
		3	3000	173

Since there are no benchmark instances and the optimal values are not known for all these instances, to measure the performance of our proposed algorithm, an upper bound is obtained by using LP-solver and the bound gap is calculated as follows:

$$Bound\ Gap = \frac{z_{bound} - z_{heu}}{z_{bound}} \tag{34}$$

The bound gap includes the heuristic gap and LP gap as shown in Figure 11. The LP gap declines with the increasing integrality index and becomes zero with full integrality.

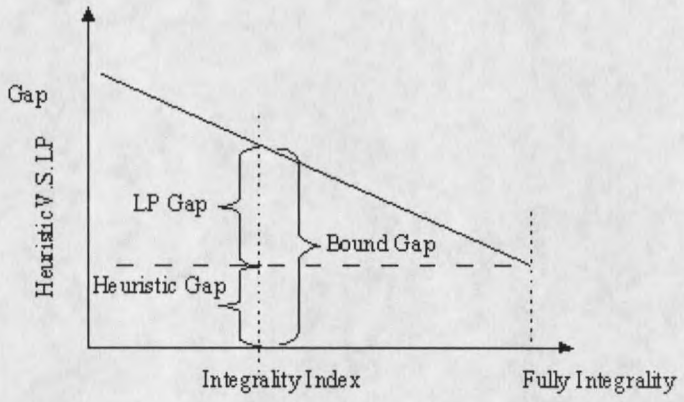

Figure 11. Heuristic Gap V.S. LP Gap

Some instances with the different horizon, the number of tasks and resource strength have been generated and tested to analyze the LP gap and Heuristic gap. Table 8 shows that the MIP running time decreases with increasing *RS* and Integrality Index. We can always get the integer LP relaxation when *RS* is set to 1 because all tasks can be scheduled. Since the bound gap is comprised of heuristic gap and LP gap, the drawback of using bound gap to measure the performance of heuristic is that we always get big LP relaxation gaps for "hard" problem. For example, for an instance of T=20, RS=0.1, the LP

relaxation gap is 21.18% while the actual heuristic gap is only 6.06%. Another reason that we may get higher gap is because of small objective value. For example, with instance of T=10, RS=0.1, the bound gap is 36.14%, the heuristic gap is 16.67% even though the result of LP relaxation is 7.83, the MIP optimal result is 6 and the heuristic results is 5.

Table 8. Results for Gap Analysis

RS	LP	MIP	MIP Running Time	Heuristic	Integrality Index
		T=100	n_i=50 5000 variables		
1.0	50	50		50	100
0.9	50	50		50	99.92
0.8	50	50	2.4 sec.	50	99.88
0.5	50	50	4 sec.	50	99.72
0.25	50	50	6.28 sec.	50	99.28
0.1	50	49*	3 hrs.	50	98.74
		T=50	n_i=50 2500 variables		
1.0	50	50		50	100
0.9	50	50	1.15 sec.	50	99.84
0.5	50	50	1.82 sec.	50	99.44
0.25	50	50	3.13 sec.	50	98.56
0.1	50	47	7 hrs	45(3hrs)	97.24
		T=20	n_i=50 1000 variables		
1.0	50	50	0.14sec.	50	100
0.9	50	50	0.46sec.	50	99.6
0.5	50	50	0.80sec.	50	98.6
0.25	50	50	1.50sec.	50	96.4
0.1	39.33	33	7 hrs	31(3hrs)	90.9
			T=10 n_i=10		
0.1	7.83	6	3hrs	5	75

* Terminated before optimal.

Average gaps for 5 replications of 60 cases are shown in Table 9. All 300 instances were run for 30 seconds on a 1.6 GHz Athelon. The first column of Table 9 is the planning horizon, T, with two levels of 10 and 20. The second column is the resource strength RS with two levels of 0.1 and 0.25. For the instances of only one mode for each task (n_{m_i}=1), the bound gap decreases drastically from 24.27% to 12.13% as *RS* increases from 0.1 to 0.25. It is also true for instances with n_{m_i}=2 and n_{m_i}=3. If everything else remains the same, the bound gap increases with the increase of the number of tasks and decreases with the planning horizon.

Table 9. Results for Computational Experiment

T	RS	Problem set n_i					Bound Gap	
		10	20	30	40	50		
n_{m_i}=1								
10	0.1	27.29	28.19	23.51	24.38	23.83	25.44	24.27
20		21.9	24.38	20.95	24.36	23.89	23.10	
10	0.25	14.28	17.24	17.32	20.39	22.54	18.35	12.13
20		5.72	7.21	2.09	9.45	5.08	5.91	
n_{m_i}=2								
10	0.1	34.64	27.35	29.18	33.03	36.02	32.04	23.26
20		13.99	9.0	17.53	14.79	17.11	14.48	
10	0.25	30.25	21.61	24.11	24.35	29.02	25.87	16.41
20		5.07	1	11.43	7.75	9.51	6.95	
n_{m_i}=3								
10	0.1	23.62	25.74	31.96	35.62	39.47	31.28	22.98
20		7.61	15.23	15.8	17.5	17.23	14.67	
10	0.25	19.62	20.64	28.71	26.79	27.88	24.73	17.95
20		3.61	10.55	11.79	15.50	14.40	11.17	

The remainder of this chapter summarizes results by factor. As shown in Figure 12, the resource strength (*RS*) has an inverse impact on the computation. The higher *RS* is, the easier the corresponding problem instance. The average gap from the simplex solution decreased from 23.5% to 15.5% when *RS* changed from 0.1 to 0.25. Based on the earlier gap analysis, the gap reduction is in part is because of a higher *RS* giving a higher Integrality Index.

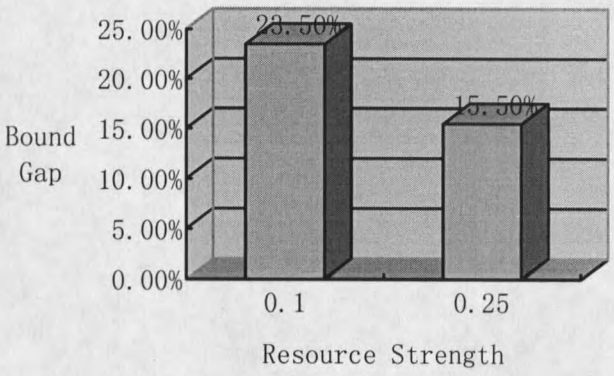

Figure 12. Impacts of *RS*

As shown in Figure 13, planning horizon has negative impact on the results. The average gap dropped from 26.29% to 11.05% when Horizon changed from 10 to 20.

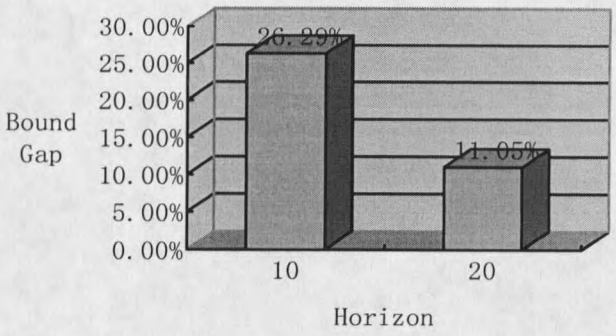

Figure 13. Impacts of Horizon

Referring to Figure 14, the number of tasks has positive impact. The average gap increases from 17.3% to 19.53% and 22.17% when the number of tasks changed from 10 to 30 and 50 respectively. As mentioned in Chapter 6, the running time per iteration is linear with problem size for inserts and quadratic with swaps, so the gap increases here are probably because of not having enough iterations to evaluate the moves for bigger problems.

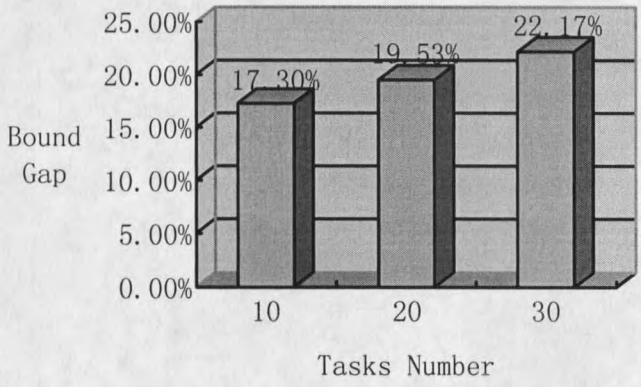

Figure 14. Impacts of Number of Tasks

The impact of the number of modes is hard to determine. On the one hand, more modes will increase the chances of finding feasible mode assignments, and on the other hand, the neighborhood used in the local search methods expands substantially and it takes a longer time for searching the neighborhood with a less thorough examination of the solution space resulting.

As shown in Figure 15 and Table 10, if we increase the running time, the average gap will decrease. Extending running time to 1 minute for the instances with n_{m_i}=1 *NR*=3, decreases the average gap from 24.27% to 23.78% for *RS*=0.1 and from 12.13% to 11.62% for *RS*=0.25.

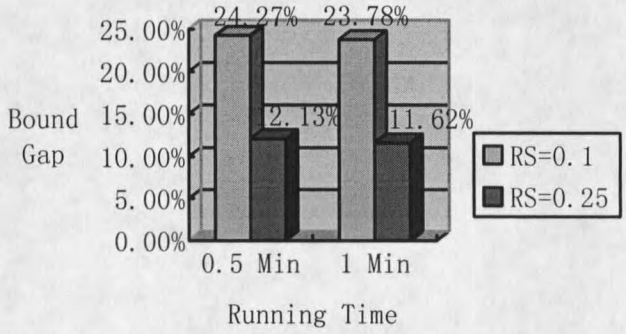

Figure 15. Impacts of Running Time

Table 10. Results for instances with n_{m_i}=1 *NR*=3

T	*RS*	Problem set n_i					Average ratio	
		10	20	30	40	50		
10	0.1	27.29	28.19	21.65	24.38	23.2	24.942	23.78
20		21.9	24.38	20.21	23.75	22.88	22.624	
10	0.25	14.28	17.24	16.65	19.84	21.22	17.846	11.62
20		5.72	7.21	1.43	7.88	4.68	5.384	

CHAPTER 8

CONCLUSIONS

In this paper a multiple parallel machines scheduling problem with setup resources (MPMSRP) in which a set of independent tasks need to be processed on a set of renewable and nonrenewable resources is modeled and studied. Each task can be carried out in multiple alternative modes, or resource sets, each with a different processing time. The objective is to assign a mode and a start time for each task so that the throughput is maximized.

Three points need to be addressed here. First, by surveying the literature, we found MPMSRP was seldom studied, especially with the additional setup resource constraints and multiple modes considered in our model. MPMSRP has wide practical usage, such as setup operators that can setup different machines, railcar loading, etc.

Second, by doing a preliminary property study, we showed that MPMSRP does not have the integrality property in general, but one of the interesting findings is the impact of setup resource constraints on integrality and solution difficulty. It was proved that when setup resource constraints become tighter, the chances of getting integer results with the LP relaxation of our proposed model were increased and solution with the LP MIP solver is easier.

Third, since MPMSRP is NP-hard, we presented a local search heuristic. The computational experiments on a set of generated problems were presented and the heuristic performance was measured by comparing with LP results. Our algorithm is conceptually elegant and straightforward to implement. As such, we expect that it may be of practical value in several areas.

Further research effort in this area will be devoted to the improvement of the proposed algorithm by means of its evaluation with respect to lower bounds of the problem. Tabu search or other local search heuristics may be implemented and evaluated to compare with the proposed algorithms.

REFERENCES CITED

[1] Li, F., *A multi-period, dynamic asphalt railcar scheduler.* IME575 Professional Paper & Project, Montana State University, 1999.

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

[3] Ventura, J.A. and D. Kim, *Parallel machine scheduling about an unrestricted due date and additional resource constraints.* IIE Transactions, 2000. **32**: p. 147-153.

[4] Blazewicz, J., et al., *Scheduling under resource constraints-deterministic models.* Annals of Operations Research, 1986. **7**.

[5] Slowinski, R., *Production Scheduling on Parallel Machines Subject to Staircase Demands.* Engineering Costs and Production Economics, 1987. **14**: p. 11-17.

[6] J.W.Herrmann and C-Y.Lee, *On parallel-machine scheduling with operator-constrained setups.* Technical Research Report, 1994: p. 85.

[7] Olafsson, S. and L. Shi, *A method for scheduling in parallel manufacturing systems with flexible resources.* IIE Transactions, 2000. **32**.

[8] Ya Yang, S.K.a.M.P., *Heuristics for minimizing total wighted tardiness in flexible flow shops.* Journal of scheduling, 2000. **3**: p. 89-108.

[9] Chung Y.L. , S.C.C., *Scheduling flexible flow shops with sequence-dependent setup effects.* IEEE TRANSACTIONS ON ROBOTICS AND AUTOMATION, 2000. **16**: p. 408-419.

[10] Daniels and Mazzola, *Flowshop Scheduling with Resource Flexibility.* Operations Research Society of America, 1994. **42**: p. 504-522.

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

[12] D.Sherali, H., S.M. .Al-Yakoob, and M. Hassan, *Fleet management models and algorithms for an oil-tanker routing and scheduling problem.* IIE Transaction, 1999. **31**: p. 395-406.

[13] Powell, W.B., J.A. Shapiro, and H.P. Simao, *Dynamic Management of Heterogenous Resources.* Statistics and Operations Research, 1998(Technical Report SOR-98-06.).

[14] Kelly, J.X.a.J.P., *A network flow-based tabu search heuristic for the vehicle routing problem.* Transportation Science, 1996. **30**(4).

[15] Salewski, Schirmer, and Drexl, *Project Scheduling under Resource and Mode Identity Constraints.* European Journal of Operational Research, 1997.

[16] Shewchuk, J.P. and Chang, *Resource-constrained job scheduling with recyclable resources European Journal of Operational Research.* 1995. **81**: p. 364-375.

[17] Reyck, B.D. and E. Herroelen, *The multi-mode resource-constrained project scheduling problem with generalized precedence relations.* European Journal of Operational Research, 1999. **119**: p. 538-556.

[18] Drexl, R.K.a.A., *Local search for nonpreemptive multi-mode resource-constrained project scheduling.* IIE Transactions, 1997. **29**: p. 987-999.

[19] Pritsker, A.A.B., L.J. Watters, and P.M. Wolfe, *Multiproject scheduling with limited resources: A zero-one programming approach.* Management Science, 1969. **16**(1): p. 93-108.

[20] Kolisch and Sprecher, *PSPLIB- A project scheduling problem library.* European Journal of Operational Research, 1996. **96**: p. 205-216.

[21] Pascoe, T.L., *"Allocation of resources-CPM"*. Revue Francaise de Recherche Operationelle, 1966. **38**: p. 31-38.

[22] Bianco, Dell, and Speranza, *Scheduling independent tasks with multiple modes*. Discrete Applied Mathematics, 1995. **62**: p. 35-50.

APPENDICES

APPENDIX A

GENERATOR AND LOCAL SEARCH SOURCE CODE

util.h

```
int max(int x, int y);
int min(int x, int y);
float u16807d(long *iseed);
int Uniform(long *iseed, int lowb, int highb);
double CpuTime(void);
char *ClockTime( float ss );
float Rnd( long *seed );
```

util.cpp

```
#include "util.h"
#include "math.h"
#include <ctype.h>
#include <time.h>
#include <string.h>
#include <stdio.h>

int max(int x, int y)
{
      if (x >= y)
      {
            return x;
      }
      return y;
}

int min(int x, int y)
{
      if (x <= y)
      {
            return x;
      }
      return y;
}


/* Random Number Generator */
float u16807d(long *iseed)
{
     *iseed = (int) fmod(*iseed * 16807.0, 2147483647.);
     return (float) ( *iseed / 2147483648.);
} /* Random Number Generator End */

int    Uniform(long *iseed, int lowb,int highb)
{
     return (int ) (lowb + u16807d(iseed) * (highb-lowb+1));
}
/* ---------------------------------------------------------- ClockTime()
   Description:
      Converts internal clock time to string hh:mm:ss.ss format
   Revisions:
      9/7/90, created .. elm
```

```
*/
char *ClockTime( float ss )
{
 int hh, mm;
 static char time[15] = "00:00:00.00";

  hh =(int) ss / 3600;
  ss =(float) fmod( ss, 3600.0 );
  mm = (int) ss / 60;
  ss = (float) fmod( ss, 60.0);
  sprintf( time, "%02d:%02d:%05.2f", hh, mm, ss );

  return time;

} /* end ClockTime() */


/* ------------------------------------------------------------ CpuTime()
   Description:
     Returns cpu time in seconds as a floating point number
   Revisions:
     2/6/91, created .. elm
*/

double CpuTime(void)
{

 return (double) clock() / (double) CLOCKS_PER_SEC; /* may need this CLK_TCK; */


} /* end CpuTime() */

/* ------------------------------------------------------------------ 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() */
```

Mode.h

```
#ifndef _MODES_H
#define _MODES_H
#include <vector>
using namespace std;

class Mode
{
public:
    int Duration;       //    duration for task i in mode m
    vector<int> *R1_Req;    //request per period for parallel,renewable resources
    vector<int> *R2_Req;    //request per period for setup, renewable resources
    vector<int> *R3_Req;    //request per period for setup, consumable resources
    Mode();
    virtual ~Mode();
};

class ModeArray
{
public:
    int ModesCount;                         //total number of modes
    vector<Mode*> *Modes ;                  //Mode Set for the task

     int NR1,NR2,NR3;                       //number of Resource type R1,R2,R3
     vector <int> *R1Max;                   //maximal number of parallel,renewable
resources
     vector <int> *R2Max;                   //maximal number of setup, renewable
resources
    vector <int> *R3Max;                    //maximal number of setup, consumable
resources

    void Add_R1Max(int Max);                //add R1 Max to R1Max Array
     void Add_R2Max(int Max);               //add R2 Max to R2Max Array
     void Add_R3Max(int Max);               //add R3 Max to R3Max Array

    ModeArray();                            //constructor
    virtual ~ModeArray();                   //destructor
    int Get_ModesCount();                   //get tatal number of modes
    void AddMode(Mode *tmpMode);            //find the mode index
    void EmptyMode();     //empty mode array
    void SetNR(int nr1,int nr2,int nr3);    //set the number of Resource type R1,R2,R3
};

#endif
```

Mode.cpp

```
#include "Modes.h"
#include <vector>


Mode::Mode()
{
    R1_Req=new vector <int>;
    R2_Req=new vector <int>;
    R3_Req=new vector <int>;

}
Mode::~Mode()
{
    delete R1_Req;
    delete R2_Req;
    delete R3_Req;

}

int ModeArray::Get_ModesCount()
{
    return ModesCount;
}

ModeArray::ModeArray()
{

 Modes=new vector<Mode*>;   //initilize the Modes
 ModesCount=0 ;             //initilize to zero
 R1Max=new vector <int> ;
 R2Max=new vector <int> ;
 R3Max=new vector <int> ;
}

ModeArray::~ModeArray()
{
    delete R1Max;
     delete R2Max;
     delete R3Max;
    delete Modes;
}
void ModeArray::EmptyMode()
{
```

```
    Modes->erase (Modes->begin(), Modes->end() );
    ModesCount=0;
}


void ModeArray::AddMode(Mode *tmpMode)
{
    Modes->push_back(tmpMode);
    ModesCount++;
}

void ModeArray::SetNR(int nr1,int nr2,int nr3)
{
    NR1=nr1;
    NR2=nr2;
    NR3=nr3;
}

void ModeArray::Add_R1Max(int Max)
{
    R1Max->push_back(Max);
}

void ModeArray::Add_R2Max(int Max)
{
    R2Max->push_back(Max);
}
void ModeArray::Add_R3Max(int Max)
{
    R3Max->push_back(Max);
}
```

Tasks.h

```
#ifndef _TASKS_H
#define _TASKS_H

#include "Modes.h"
#include "Slots.h"
#include <vector>
#include <list>
using namespace std;

struct BaseStruct
{
      long   Seed          ; //seed
      int   Horizon       ; // horizon
       int   NrOfTasks     ; // # Taskss
       int   MinMode        ; // min #modes per job
       int   MaxMode        ; // max #modes per job
       int   MinDur         ; // min duration
       int   MaxDur         ; // max duration

       int   MinR1          ; // min # R1 resources
       int   MaxR1          ; // max # R1 resources
       int   MinR1Req       ; // min period request
       int   MaxR1Req       ; // max period request
       float R1RF           ; // res. factor ren. res.
       float R1RS           ; // res. strength ren. res.

      int   MinR2          ; // min # R2 resources
       int   MaxR2          ; // max # R2 resources
       int   MinR2Req       ; // min period request
       int   MaxR2Req       ; // max period request
      float R2RF           ; // res. factor ren. res.
       float R2RS           ; // res. strength ren. res.

       int   MinR3          ; // min # R3 resources
       int   MaxR3          ; // max # R3 resources
       int   MinR3Req       ; // min period request
       int   MaxR3Req       ; // max period request
      float R3RF           ; // res. factor ren. res.
       float R3RS           ; // res. strength ren. res.

       //float Prob          ; // prob. of res. mode desity
};
```

```
class Task
{
public:
    int     NrOfModes;       // number of modes for the task
    vector<Mode*> *ModeSet;     //Mode set for the job
     Task();
    virtual ~Task();
};


class TaskArray
{
private:



    char dName[48];                    // problem data file (probnam.dat)
    char baseName[48];                 // base data file (base.dat)
    int   TaskCount;                              //the number of tasks
     int Horizon;              // maximal horizon
     int Seed;                 //seed of test instance
    vector <Task*> *Tasks;   //the array of tasks



public:
    ModeArray *TasksModes;                          //Tasks Modes
     BaseStruct Base;                      //base date for generate tasks data

     void SetDataFileName(char *FileName);    //Set the data filename
    void SetBaseFileName(char *fName) ;     //Set the base data filename
    int   Get_TaskCount();                    //get the total number of tasks
    void Set_TaskCount(int Count);          //set the total number of tasks
     int   Get_NR(int Rtype);                          //get the number of resource type
Rtype
    int   Get_RMax(int RType,int index);             //get the resource capacity of
Resource type R1,R2,R3


    void Set_NR(int nr1,int nr2,int nr3); //set the number of resource type
                                                  // R1-parallel,renewable resources
                                                   // R2-setup,renewable   resources
                                                   // R3-setup,consumable resources
```

```
    void Set_Horizon(int T);                    //set the horizon
    int   Get_Horizon();                        //get the horizon
    void Set_Seed(long s);                      //set the seed
     TaskArray();                                        //constructor
    virtual ~TaskArray();                               //destructor
    void AddTask(Task *newtask);     //add a task to index i of taskarray
    void EmptyTask();   //empty the task array
    void Incr_TaskCount();                      //increase TaskCound()
    void ReadTasks();                   //read data from data file
    void WriteTasks(char outName[]);                    //write data to the disk
    void GeneTasks();                   //generate tasks instance
    void InputBase();                   //read data from base file
    Task *GetTask(int index);   //get task from task array by index
    int   Get_NrOfModes(int i);   //get the number of modes of task i
    void Print() ;                  //print tasks information
    int   Determine(int element,int ResourceType);     //determines the randomly chosen
tripel [i,m,r]
    int   MinUseR1(int r1) ;      // returns the Max (minimum usage for resource r of
Resource Type R1)
    int   MaxUseR1(int r1) ;      // returns the Max period use of Resource Type R1)
    int   MinUseR2(int r2) ;      // returns the Max (minimum usage for resource r of
Resource Type R2)
    int   MaxUseR2(int r2) ;      // returns the Max period use of Resource Type R2)
     int   MinUseR3(int r2) ;      // returns the Max (minimum usage for resource r of
Resource Type R3)
    int   MaxUseR3(int r2) ;      // returns the Max period use of Resource Type R3)
    int   Get_R_Req(Mode *ModeTmp,int RType,int rindex);   //get the usage for
mode,resouce type Rtype and r index.
    void   Set_R_Req(Mode *ModeTmp,int RType,int rindex,int Req);   //set the usage
for mode,resouce type Rtype and r index.
};

#endif
```

Slots.h

```
#ifndef _SLOTS_H
#define _SLOTS_H
#include "Tasks.h"
#include <vector>
#include <list>
using namespace std;

class Slot
{
public:
    Mode   *M;     //   resources R1,R2,R3
    int      T;     //   time T
     list <int>   TaskSet; // tasks set for the slot    index i
    Slot();
    virtual ~Slot();
};

class SlotArray
{
private:

    vector <Slot*> *Slots ;    // all the (r,t) pair

public:
     int SlotCount;
     SlotArray();                    //constructor
    virtual ~SlotArray();      //destructor
    void AddSlot(Slot *tmp);    //add slot to the Slots
    int GetSlotCount();             //get the slot count
    Mode *GetMode(int s);      //get the mode of the slot s
    int GetT(int s);
};

#endif
```

Slots.cpp

```
#include "Slots.h"

Slot::Slot()
{
M=new Mode;
}

Slot::~Slot()
{
delete M;
}

void SlotArray::AddSlot(Slot *tmp)
{
     Slots->push_back(tmp);
      SlotCount++;
}

SlotArray::SlotArray()       //constructor
{
    Slots=new vector <Slot*>; //initilize Slots
    SlotCount=0;
}

SlotArray::~SlotArray()       //destructor
{
     delete Slots;     //free memory
}

int SlotArray::GetSlotCount()     //get slot count
{
     return SlotCount;
}

Mode *SlotArray::GetMode(int s)     //get the mode of the slot s
{
     return (*Slots)[s]->M;
}

int SlotArray::GetT(int s)                //get t(s)
{
     return (*Slots)[s]->T;
}
```

Choices.h

```
#ifndef _CHOICES_H
#define _CHOICES_H
#include "Tasks.h"
#include "Slots.h"
#include "util.h"

#include <vector>
#include <list>
using namespace std;

#define TRUE 1
#define FALSE 0

static const float Penalty = 100 ;

struct Choice              //   j=( i,r,t)
{
    int TaskIndex;     //     task i
    int SlotIndex;     //     slot s(r,t) pair
};
struct LHS_Struct
{
    vector <float> R1;
    vector <float> R2;
    vector <float> R3;
};
class Soln{
public:
    float z;     //objective function value
     vector<int> X;   //Xj
    vector<LHS_Struct> LHS;
    vector<int> SchTasks;    //scheduled tasks    A
    vector<int> UnSchTasks; //unscheduled tasks    U
     void CreatLHS(int Horizon,int NR1,int NR2,int NR3); //creat LHS array to track
LHS of the constraints
    int ConstViol    ; //total constraints violation
    Soln();
    virtual ~Soln();
} ;

struct Srch {                    // Search parameters and status
   long   iterbest;              // iteration when best found */
   long   insertiter;            // insert try number
```

```
    long    insertmoves;            // insert moves
    long    swapiter ;              // swap evaluation number
    long    feasibleswap;           // total feasible swaps
    long    CycleBest;              // multistart cycle
    long    TotalCycle;             //Total Cycle
    double tbest;                   // time (in seconds) when best solution found */

} ;

class   ChoiceArray
{
private:
     vector <Choice*> *Choices;     //   (i,r,t) Sets

public:

     TaskArray          *Tasks_i;    //   tasks i
     SlotArray          *Slots_s;    //   slots s
     int ChoiceCount;
     long seed;                 //seed for generate solution
     ChoiceArray();                 //constructor
     virtual ~ChoiceArray();        //destructor
     void CreateSlots();                //create slots according the modes and horizon
     void GenChoices();             //generate choices
     Choice *GetChoice(int j) ;  //get Choice j
     int    GetTaskIndex(int j);    //get the task index of choice j
     int    GetTaskCount();         //get tasks count
     int    GetSlotCount();         //get the slots count
     void AddChoice(Choice *tmp);   //add choice
     Task *GetTask(int i);              //get task i
     Mode *GetMode(int j);          //get the mode of choice j
     int    GetT(int j);            //get t of choice j
     int    Get_dm(int j);          //get the duration of choice j
      int    GetRCount(int RType);           //get resouces number of type R1,R2,R3

     int GetRMax(int RType,int index);      //get the resources capacity of index r for
resource type R1,R2,R3

     int GetHorizon();              //get horizon T

     //----------------------solution part-----------------------------
       Soln currsoln, bestsoln;// current, best solutions
       Srch localsrch;
      void Z(Soln *soln);     //get the objective value for a solution
      void Shift(Soln *soln);   //shift the tasks to reduce Cmax
```

```
        void InsertChoice(Soln *soln,int j);      //Add a choice to the solution
        int    EvalueInsert(Soln *soln,int j);    //Add a choice and get z value
        void RemoveChoice(Soln *soln,int j);      //Add a choice to the solution
        int    EvalueSwap(Soln soln,int i,int j,float *maxmin);   //swap between scheduled
task and unscheduled task

        void AssignTask(Soln *soln,int i);              //subtrack task from U set and add to A
set
        void UnassignTask(Soln *soln,int i);            //subtrack task from A set and add to
U set
        void InitSoln( Soln *soln,long *seed );
        void CopySoln( Soln *soln1,Soln *soln2); //copy soln1 to soln2 for backup
        void Search(Srch *srch,Soln *soln,long *seed);        //    local search
         float EvalueLHS(Soln *soln);
        void MultiStart(Srch *srch,Soln *soln,long *seed,int TimeLimit);

         void PrintSoln(Soln *soln);              //print out the solution
         int Get_ar1(int r1,int j);   //get resource R1 require units for that r and j
         int Get_ar2(int r2,int j);   //get resource R2 require units for that r and j
         int Get_ar3(int r3,int j);   //get resource R3 require units for that r and j

         vector <int> *Get_X_Jr1_List(int r1,Soln *soln)   ;//get the Xj list of resource r/R1
type
        vector <int> *Get_X_Jr2_List(int r2,Soln *soln)   ;//get the Xj list of resource r/R2
type
        vector <int> *Get_X_Jr3_List(int r3,Soln *soln)   ;//get the Xj list of resource r/R3
type
        vector <int> *Get_Choice_List(int i);             // get the Choice index list of task i
        float Rnd( long *seed );                           // uniform (0,1) random number
generator
};


#endif
```

Choices.cpp

```
#include "Choices.h"
#include <list>
#include "math.h"
#include <stdio.h>
#include <iostream>
#include "util.h"
#define DEBUG FALSE
Soln::~Soln()
{
}

Soln::Soln()
{
    z=0;
    ConstViol=0;
}


ChoiceArray::ChoiceArray()      //constructor
{
  Tasks_i=new TaskArray ;     //initilize the tasks and slots class
  Slots_s=new SlotArray ;
  Choices=new vector <Choice*>;
  ChoiceCount=0;
  seed=123987;
}

ChoiceArray::~ChoiceArray()   //destructor
{
    delete Slots_s;
    delete Tasks_i;              //free the memory
}

void ChoiceArray::CreateSlots()   //create slots according Modes and horizon( r,t) pair
{
    Slot *SlotTmp;
    for(int t=0;t<Tasks_i->Get_Horizon();t++)
        for(int s=0;s<Tasks_i->TasksModes->Get_ModesCount();s++)
        {
            SlotTmp=new Slot;
            SlotTmp->M=(*Tasks_i->TasksModes->Modes)[s];
            SlotTmp->T=t;
            Slots_s->AddSlot(SlotTmp);
```

```
        }

}

int ChoiceArray::GetTaskCount()
{
    return Tasks_i->Get_TaskCount();
}

int ChoiceArray::GetHorizon()
{
    return Tasks_i->Get_Horizon();
}


int ChoiceArray::GetSlotCount()
{
    return Slots_s->GetSlotCount();
}


Task * ChoiceArray::GetTask(int i)
{
    return Tasks_i->GetTask(i);
}

void ChoiceArray::AddChoice(Choice *tmp)
{
    Choices->push_back(tmp);

    ChoiceCount++;
}

void ChoiceArray::GenChoices()     //generate choices
{
    Mode *ModeTmp=new Mode;
    Task *TaskTmp=new Task;
    Choice *ChoiceTmp;

    for(int i=0;i<GetTaskCount();i++)
    {
        TaskTmp=GetTask(i);
        for(int m=0;m<TaskTmp->NrOfModes;m++)
        {
```

```
                ModeTmp=(*TaskTmp->ModeSet)[m];
                for(int s=0;s<GetSlotCount();s++)
                  {
                    if (Slots_s->GetMode(s)==ModeTmp)
                    {
                        ChoiceTmp=new Choice;
                        ChoiceTmp->TaskIndex=i;
                        ChoiceTmp->SlotIndex=s;
                        AddChoice(ChoiceTmp);
                    }

                }
            }

    }

}

int ChoiceArray::GetRCount(int Rtype)
{
    return Tasks_i->Get_NR(Rtype);
}



int ChoiceArray::Get_ar1(int r1,int j) //get resource require units for that r and j
{
    int stmp=(*Choices)[j]->SlotIndex;
    return (*(Slots_s->GetMode(stmp))->R1_Req)[r1];
}

int ChoiceArray::Get_ar2(int r2,int j) //get resource require units for that r and j
{
    int stmp=(*Choices)[j]->SlotIndex;

    return (*(Slots_s->GetMode(stmp))->R2_Req)[r2];
}

int ChoiceArray::Get_ar3(int r3,int j) //get resource require units for that r and j
{
    int stmp=(*Choices)[j]->SlotIndex;

    return (*(Slots_s->GetMode(stmp))->R3_Req)[r3];
}
```

```
int ChoiceArray::GetT(int j)
{
     int slottmp;
     slottmp=(*Choices)[j]->SlotIndex;
     return Slots_s->GetT(slottmp);
}

int ChoiceArray::GetTaskIndex(int j)
{
    return (*Choices)[j]->TaskIndex;
}

int ChoiceArray::Get_dm(int j)
{
     int stmp=(*Choices)[j]->SlotIndex;

     return    (Slots_s->GetMode(stmp))->Duration;


}

Mode *ChoiceArray::GetMode(int j)
{
      int stmp=(*Choices)[j]->SlotIndex;

     return Slots_s->GetMode(stmp);
}

int ChoiceArray::GetRMax(int RType,int index)
{
    return Tasks_i->Get_RMax(RType,index);
}



void ChoiceArray::InsertChoice(Soln *soln,int j_add)     //Add choice to solution and get
z value
{
     int r,t,t_ins,tstart,tend;
     int ar_consum=0;
     int a_r,AR1,AR2,AR3;
      int LHSVALUE;
     int horizon=GetHorizon();
     for(r=0;r<GetRCount(1);r++)     //   R1 resource constraints
```

```
{
    AR1=GetRMax(1,r);
    a_r=Get_ar1(r,j_add);
    if (a_r==0) continue ;   //skip the choice if the resource have no usage
     tstart=GetT(j_add);
    tend=min(tstart+Get_dm(j_add),horizon-1);
    for(t=tstart;t<=tend;t++)       //only consider the time period that added choice
has effection
    {

        LHSVALUE=soln->LHS[t].R1[r]+a_r;

        soln->LHS[horizon].R1[r]=soln->LHS[horizon].R1[r]+a_r;

        if (soln->LHS[t].R1[r]<AR1 && LHSVALUE>AR1)
            soln->ConstViol=soln->ConstViol+max(0,LHSVALUE-AR1);
        if (soln->LHS[t].R1[r]>=AR1)
            soln->ConstViol=soln->ConstViol+a_r;

         soln->LHS[t].R1[r]=LHSVALUE;

    }

}                   //end of R1 resource constraints
//-------------------------------------------------
    for(r=0;r<GetRCount(2);r++)     //   R2 resource constraints
{
    AR2=GetRMax(2,r);
    a_r=Get_ar2(r,j_add);
    if (a_r==0) continue ;   //skip the choice if the resource have no usage
    t=GetT(j_add);

    LHSVALUE=soln->LHS[t].R2[r]+a_r;
     soln->LHS[horizon].R2[r]=soln->LHS[horizon].R2[r]+a_r;

    if (soln->LHS[t].R2[r]<AR2 && LHSVALUE>AR2)
        soln->ConstViol=soln->ConstViol+max(0,LHSVALUE-AR2);
     if (soln->LHS[t].R2[r]>=AR2 )
        soln->ConstViol=soln->ConstViol+a_r;
     soln->LHS[t].R2[r]=LHSVALUE;


}                   //end of R2 resource constraints
//-------------------------------------------------
for(r=0;r<GetRCount(3);r++)     //   R3 resource constraints
```

```
    {
        AR3=GetRMax(3,r);
        a_r=Get_ar3(r,j_add);
        if (a_r==0) continue ;    //skip the choice if the resource have no usage
         ar_consum=0;
        t_ins=GetT(j_add);

        soln->LHS[t_ins].R3[r]=soln->LHS[t_ins].R3[r]+a_r;

        if (soln->LHS[horizon].R3[r]>=AR3)
                soln->ConstViol=soln->ConstViol+a_r;
         if (soln->LHS[horizon].R3[r]<AR3 && soln->LHS[horizon].R3[r]+a_r>AR3 )

soln->ConstViol=soln->ConstViol+max(0,soln->LHS[horizon].R3[r]+a_r-AR3);

        soln->LHS[horizon].R3[r]=soln->LHS[horizon].R3[r]+a_r;

    }                   //end of R3 resource constraints
     soln->X.push_back(j_add);
    soln->z=soln->X.size()-soln->ConstViol*Penalty;

}


int ChoiceArray::EvalueInsert(Soln *soln,int j_add)     //Add choice to solution and get z
value
{
    int r,t,ztmp;
    int tstart,tend;
    int ar_consum=0;
    int a_r,AR1,AR2,AR3;

     int       Violation=soln->ConstViol;
    int horizon=GetHorizon();
    for(r=0;r<GetRCount(1);r++)     //    R1 resource constraints
    {
        AR1=GetRMax(1,r);
        a_r=Get_ar1(r,j_add);
        if (a_r==0) continue ;    //skip the choice if the resource have no usage
          tstart=GetT(j_add);
        tend=min(tstart+Get_dm(j_add),horizon-1);
        for(t=tstart;t<=tend;t++)     //only consider the time period that added choice has
effection
        {
```

```
        ar_consum=soln->LHS[t].R1[r]+a_r;

        if (soln->LHS[t].R1[r]<AR1 && ar_consum>AR1)
            Violation=Violation+max(0,ar_consum-AR1);
        if (soln->LHS[t].R1[r]>=AR1)
            Violation=Violation+a_r;

    }

}               //end of R1 resource constraints
//-------------------------------------------------
    for(r=0;r<GetRCount(2);r++)     //   R2 resource constraints
{
    AR2=GetRMax(2,r);
    a_r=Get_ar2(r,j_add);
    if (a_r==0) continue ;   //skip the choice if the resource have no usage
    t=GetT(j_add);
    ar_consum=soln->LHS[t].R2[r]+a_r;


    if (soln->LHS[t].R2[r]<AR2 && ar_consum>AR2)
            Violation=Violation+max(0,ar_consum-AR2);
    if (soln->LHS[t].R2[r]>=AR2)
            Violation=Violation+a_r;

}               //end of R2 resource constraints
//-------------------------------------------------
for(r=0;r<GetRCount(3);r++)     //   R3 resource constraints
{
    AR3=GetRMax(3,r);
    a_r=Get_ar3(r,j_add);
    if (a_r==0) continue ;   //skip the choice if the resource have no usage

    ar_consum=soln->LHS[horizon].R3[r]+a_r;

    if (soln->LHS[horizon].R3[r]>=AR3)
          Violation=Violation+a_r;

     if (soln->LHS[horizon].R3[r]<AR3 && ar_consum>AR3 )
            Violation=Violation+max(0,ar_consum-AR3);

}               //end of R3 resource constraints

ztmp=soln->X.size()+1-Violation*Penalty;
```

```
    return ztmp;
}

void ChoiceArray::RemoveChoice(Soln *soln,int j_sub)     //Remove a choice from
solution and get z value
{
    int r,t,t_sub,tstart,tend;
    int ar_consum=0;
    int a_r,AR1,AR2,AR3;
     int LHSVALUE;
    int horizon=GetHorizon();
    for(r=0;r<GetRCount(1);r++)     //   R1 resource constraints
    {
        AR1=GetRMax(1,r);
        a_r=Get_ar1(r,j_sub);
        if (a_r==0) continue ;   //skip the choice if the resource have no usage
          tstart=GetT(j_sub);
        tend=min(tstart+Get_dm(j_sub),horizon-1);
        for(t=tstart;t<=tend;t++)     //only consider the time period that added choice has
effection
        {

            LHSVALUE=soln->LHS[t].R1[r]-a_r;
            soln->LHS[horizon].R1[r]=soln->LHS[horizon].R1[r]-a_r;

            if ( LHSVALUE >=AR1)
                soln->ConstViol=soln->ConstViol-a_r;

            if (soln->LHS[t].R1[r]>AR1 && LHSVALUE<AR1)
                soln->ConstViol=soln->ConstViol-(soln->LHS[t].R1[r]-AR1);

             soln->LHS[t].R1[r]=LHSVALUE;

        }

    }                 //end of R1 resource constraints
    //-------------------------------------------------
        for(r=0;r<GetRCount(2);r++)     //   R2 resource constraints
    {
        AR2=GetRMax(2,r);
        a_r=Get_ar2(r,j_sub);
        if (a_r==0) continue ;   //skip the choice if the resource have no usage
        t=GetT(j_sub);

        LHSVALUE=soln->LHS[t].R2[r]-a_r;
```

```
    soln->LHS[horizon].R2[r]=soln->LHS[horizon].R2[r]-a_r;

     if ( LHSVALUE >=AR2)
            soln->ConstViol=soln->ConstViol-a_r;

    if (soln->LHS[t].R2[r]>AR2 && LHSVALUE<AR2)
        soln->ConstViol=soln->ConstViol-(soln->LHS[t].R2[r]-AR2);

     soln->LHS[t].R2[r]=LHSVALUE;


}                   //end of R2 resource constraints
//-----------------------------------------------
for(r=0;r<GetRCount(3);r++)     //   R3 resource constraints
{
    AR3=GetRMax(3,r);
    a_r=Get_ar3(r,j_sub);
    if (a_r==0) continue ;   //skip the choice if the resource have no usage
     ar_consum=0;
    t_sub=GetT(j_sub);

     ar_consum=soln->LHS[horizon].R3[r]-a_r;

    if (ar_consum>=AR3 )
        soln->ConstViol=soln->ConstViol-a_r;

    if (soln->LHS[horizon].R3[r]>AR3 && ar_consum<AR3 )
        soln->ConstViol=soln->ConstViol-(soln->LHS[horizon].R3[r]-AR3);


    soln->LHS[t_sub].R3[r]=soln->LHS[t_sub].R3[r]-a_r;

}                   //end of R3 resource constraints

vector<int>::iterator first = soln->X.begin();
 vector<int>::iterator last   = soln->X.end();

 while (first != last)
{
    if (*first==j_sub)
        break;
    else
        first++;
}
```

```
        soln->X.erase(first);

        soln->z=soln->X.size()-soln->ConstViol*Penalty;

}

vector <int> * ChoiceArray::Get_X_Jr1_List(int r1,Soln *soln)   //get the Xj list of
resource r type
{
    int a_r;
    vector <int> *listtmp;
     listtmp=new vector <int>;
    int jtmp,j;
     for(jtmp=0;jtmp<soln->X.size();jtmp++)
    {
            j=soln->X[jtmp];
              a_r=Get_ar1(r1,j);
            if (a_r>0)
                listtmp->push_back(j);
    }
    return listtmp;
}

vector <int> * ChoiceArray::Get_X_Jr2_List(int r2,Soln *soln)   //get the Xj list of
resource r/R2 type
{
    int a_r;
    vector <int> *listtmp;
     listtmp=new vector <int>;
    int jtmp,j;
     for(jtmp=0;jtmp<soln->X.size();jtmp++)
    {
            j=soln->X[jtmp];
              a_r=Get_ar2(r2,j);
            if (a_r>0)
                listtmp->push_back(j);
    }
    return listtmp;
}

vector <int> * ChoiceArray::Get_X_Jr3_List(int r3,Soln *soln)   //get the Xj list of
resource r/R3 type
{
    int a_r;
    vector <int> *listtmp;
```

```
    listtmp=new vector <int>;
    int jtmp,j;
    for(jtmp=0;jtmp<soln->X.size();jtmp++)
    {
            j=soln->X[jtmp];
              a_r=Get_ar3(r3,j);
            if (a_r>0)
                listtmp->push_back(j);
    }
    return listtmp;
}


Choice * ChoiceArray::GetChoice(int j)
{
    return   (*Choices)[j];
}


vector <int> * ChoiceArray::Get_Choice_List(int i)   //get choice list of task i
{
    vector <int> *ChoiceList=new vector <int>;
    Choice   *ChoiceTmp=new Choice;

    for (int j=0;j<ChoiceCount;j++)
    {
        *ChoiceTmp=*GetChoice(j);
       if (ChoiceTmp->TaskIndex==i)
            ChoiceList->push_back(j);
    }
    delete ChoiceTmp;
    return ChoiceList;
}

float ChoiceArray::EvalueLHS(Soln *soln)
{

    int horizon=GetHorizon();
    float ratio=0;
        int AR1,AR2,AR3;
        int r1,r2,r3;
    for(r1=0;r1<GetRCount(1);r1++)
    {
```

```
        AR1=GetRMax(1,r1);


        float result=(float) soln->LHS[horizon].R1[r1]/horizon/AR1;
        soln->LHS[horizon+1].R1[r1]=result;
        if (DEBUG==TRUE)      printf("LHS1=%f\n",result);
        if (result>ratio)
            ratio=result;
    }
    for(r2=0;r2<GetRCount(2);r2++)
    {
        AR2=GetRMax(2,r2);


        float result=(float) soln->LHS[horizon].R2[r2]/horizon/AR2;
        soln->LHS[horizon+1].R2[r2]=result;
        if (DEBUG==TRUE) printf("LHS2=%9.7f\n",result);
        if (result>ratio)
            ratio=result;
    }

    for(r3=0;r3<GetRCount(3);r3++)
    {
        AR3=GetRMax(3,r3);


        float result=(float) soln->LHS[horizon].R3[r3]/horizon/AR3;
        soln->LHS[horizon+1].R3[r3]=result;
      if (DEBUG==TRUE)   printf("LHS3=%f\n",result);
      if (result>ratio)
            ratio=result;
    }
    if (DEBUG==TRUE) printf("-------\n");
    return ratio;
}

void Soln::CreatLHS(int Horizon,int NR1,int NR2,int NR3) //creat LHS array to track
LHS of the constraint
{
    LHS_Struct LHSTmp;
    for(int r1=0;r1<NR1;r1++)
    LHSTmp.R1.push_back(0);
    for(int r2=0;r2<NR2;r2++)
    LHSTmp.R2.push_back(0);
    for(int r3=0;r3<NR3;r3++)
```

```
    LHSTmp.R3.push_back(0);
     for (int t=1;t<=Horizon+1;t++)
    {
       LHS.push_back(LHSTmp);
    }
      LHS.push_back(LHSTmp);    // LHS[Horizon] is used to store some ration of
resource usage
}

void ChoiceArray::InitSoln(Soln *soln,long *seed)
{
    int i,itmp,jtmp,j;


    soln->CreatLHS(GetHorizon(),GetRCount(1),GetRCount(2),GetRCount(3));

    vector <int> *ChoiceList=new vector <int>;

    bool UnAssigned=TRUE;
     int Counter=1;
    int UnSchSize=0;
    //initialize    the Assigned(A) and Unassigned(U) tasks set
    for( i=0;i<GetTaskCount();i++)
        soln->UnSchTasks.push_back(i);    //all tasks are unassigned

    for(Counter=0;Counter<=100;Counter++)
    {
        UnSchSize=soln->UnSchTasks.size();
        if (UnSchSize==0)    break;
        itmp=Uniform(seed,0,UnSchSize-1); //random assigned one task from U
        i=soln->UnSchTasks[itmp];
        *ChoiceList=*Get_Choice_List(i);

         jtmp = Uniform(seed,0,ChoiceList->size()-1);      //random select a choice for
that task i
        j=(*ChoiceList)[jtmp];

        int ztmp=EvalueInsert(soln,j);    //add a choice j to the solution to evaluate
        if (ztmp>0)
        {
        InsertChoice(soln,j);
        AssignTask(soln,i);
        }
    }
     delete ChoiceList;
```

```
}

void ChoiceArray::AssignTask(Soln *soln,int i)
{
    vector<int>::iterator first;
    vector<int>::iterator last;
    first = soln->UnSchTasks.begin();
     last   = soln->UnSchTasks.end();

     while (first != last)
     {
         if (*first==i)
              break;
         else
              first++;
     }

     soln->UnSchTasks.erase(first);//remove the task from U to A
     soln->SchTasks.push_back(i);   //add the task in the list

}

void ChoiceArray::UnassignTask(Soln *soln,int i)
{
    vector<int>::iterator first;
    vector<int>::iterator last;
    first = soln->SchTasks.begin();
     last   = soln->SchTasks.end();

     while (first != last)
     {
         if (*first==i)
              break;
         else
              first++;
     }

     soln->SchTasks.erase(first);//remove the task from A to U
     soln->UnSchTasks.push_back(i);   //add the task in the list

}

int ChoiceArray::EvalueSwap(Soln soln,int j1,int j2,float *minmax)    //swap between
scheduled task choice j1 and unscheduled task choice j2
{
```

```
    //---------------------------------------
     int ztmp;
    RemoveChoice(&soln,j1);
    InsertChoice(&soln,j2);
    ztmp=soln.X.size()-soln.ConstViol*Penalty;
     *minmax=EvalueLHS(&soln) ;

     return ztmp;
}

/*void ChoiceArray::Shift(Soln *soln)
{
   int Cmax,minstart;
   int i;
   for (i=0;i<soln->X.size();i++)
   {

   }

}
*/
void ChoiceArray::Search(Srch *srch,Soln *soln,long *seed)    //local search heuristic
{
   vector <int> *ChoiceList;
   long iteration=0;
   int counter=0;
   long swapiter = 0;              // swap iteration counter
   long insertiter = 0;            // insert iteration counter
   float zbest,minmax=1e+10,minmax2;              // save best z value
   int i,iRemove,ui,uitmp,ujtmp,uijtmp,j,uij,jtmp,ztmp;
   int jbest,uibest,uijbest;
   int UnSchSize;
   int taskindex;
   ChoiceList=new vector <int>;
   short improve = TRUE;
   short insertimprove= TRUE;
   short swapimprove=TRUE;
   short swap=FALSE;
   short insert=FALSE;
   int     maxviolation=1e+10;
   while (improve)
   {
     improve=FALSE;
     iteration++;
```

```
zbest = soln->z;       //save the z value
while (insertimprove)
{
    insert=FALSE;
    insertimprove=FALSE;
    UnSchSize=soln->UnSchTasks.size();
    if (UnSchSize==0 )     //all scheduled    exit
    {
        improve=FALSE;
        insertimprove=FALSE;
        swapimprove=FALSE;
        continue;
    }
    for(uitmp=0;uitmp<UnSchSize;uitmp++)
    {
        i=soln->UnSchTasks[uitmp];
        *ChoiceList=*Get_Choice_List(i);   //get a choice list for that task
        for( ujtmp=0;ujtmp< ChoiceList->size();ujtmp++)
        {
            srch->insertiter++;
            j=(*ChoiceList)[ujtmp];

            ztmp=EvalueInsert(soln,j);



            if (ztmp> zbest)     //if improved (greedy)
            {
                srch->tbest = CpuTime();
                srch->insertmoves++;
                insert=TRUE;
                insertimprove=TRUE;
                zbest=ztmp;       //save the best z value
                InsertChoice(soln,j);          //select this move (add the choice)
                AssignTask(soln,i);   //assign the task i
                //Shift(soln);     //try to reduce the Cmax
                minmax=EvalueLHS(soln) ;

                break;     //insert this task and go for another unassigned task
            }


        }   //end for ujtmp
        if (insert==TRUE)
        {
```

```
                    //printf("\n after insert ---------> \n");
                  //PrintSoln(soln);   //printout after insert
                  insert=FALSE;
                  break;
               }
          }   //end for uitmp
     } //end insertimprove while
/*
     InsertChoice(soln,jsave);   //insert one with least vilation

     AssignTask(soln,isave);   //assign the task i
*/

      while(swapimprove)
      {
         swap=FALSE;
         swapimprove=FALSE;
         minmax=EvalueLHS(soln);
         for(jtmp=0;jtmp<soln->X.size();jtmp++)          //assigned set A == assigned
Choice Set X
         {
              j=soln->X[jtmp];
               for(uitmp=0;uitmp<soln->UnSchTasks.size();uitmp++)   //unassigned set
U
              {
                   ui=soln->UnSchTasks[uitmp];

                    *ChoiceList=*Get_Choice_List(ui);   //get a choice list for that
unassigned task ui
                  for(uijtmp=0;uijtmp<ChoiceList->size();uijtmp++)     //each choice
for that unassigned task ui
                  {
                     srch->swapiter++;

                     uij=(*ChoiceList)[uijtmp];
                        taskindex=GetTaskIndex(uij);

                      ztmp=EvalueSwap(*soln,j,uij,&minmax2);       //swap
unscheduled with scheduled
                     if (DEBUG==TRUE) printf("evalue to swap\n");
                     if (ztmp>0)
                     {
                          srch->feasibleswap++;
                          if(minmax2<minmax)
                          {
```

```
                        swap=TRUE;
                        zbest=ztmp;       //save the best z value
                        minmax=minmax2;          //save the choice to be swaped
                        jbest=j;
                        uibest=ui;
                        uijbest=uij;
                    }

                }
            } //end for uitmp
        }// end for ui
    } //end for jtmp
    if(swap )     //got a feasible swap ???
    {
     //printf("\n before swaped ---------> \n");
      //PrintSoln(soln);    //printout before swap

    swap=FALSE;
      swapimprove=TRUE;
    swapiter++;
    srch->iterbest=swapiter;
      improve=TRUE;
    insertimprove=TRUE;    //try insert again after success swap
    iRemove=GetTaskIndex(jbest);
    // taskindex=GetTaskIndex(uijbest);
    RemoveChoice(soln,jbest);
    UnassignTask(soln,iRemove);
    //taskindex=GetTaskIndex(uijbest);
    InsertChoice(soln,uijbest);

    AssignTask(soln,uibest); //add the task to the list
    //printf("\n after swaped----------->    \n");
    //PrintSoln(soln);

    }
    if (DEBUG==TRUE) printf("after swaped\n");


  }   //end wile swap improving.

 }   //end while improve

}   //end search function
```

```
void ChoiceArray::PrintSoln(Soln *soln)
{
    int t,jtmp,j,nr1,nr2,nr3;
    Choice *Choicetmp=new Choice;
    Mode *Modetmp=new Mode;


    printf("\n Solution results \n");
     printf("Objective Function Value is Max Z=%9.2f \n",soln->z);
    printf("Constraints vialations are   %d   \n",soln->ConstViol);

    printf(" Task |Start| Duration     Resource request units   \n");
    for(jtmp=0;jtmp<soln->X.size();jtmp++)
    {
         j=soln->X[jtmp];
        Choicetmp=GetChoice(j);       //get the choice of index j
         printf("%3d",Choicetmp->TaskIndex+1);

         Modetmp=GetMode(j);     //get the mode of the choice j
        t=GetT(j)+1;                  //get the start time
        printf("      %3d     %3d     | ",t,Modetmp->Duration );

         for(nr1=0;nr1<GetRCount(1);nr1++)
           printf("%3d ",Get_ar1(nr1,j));
        for(nr2=0;nr2<GetRCount(2);nr2++)
           printf("%3d ",Get_ar2(nr2,j));
        for(nr3=0;nr3<GetRCount(3);nr3++)
           printf("%3d ",Get_ar3(nr3,j));
        printf("\n");

    }

}


void ChoiceArray::MultiStart(Srch *srch,Soln *currsoln,long *seed,int TimeLimit)
{
Soln bestsoln,tmpsoln;// tmp solutions
int zbest=0;
double tbest;
double start_time,end_time;
tmpsoln=*currsoln;
start_time = CpuTime();
```

```
end_time=start_time+TimeLimit;

while ( CpuTime()<end_time)
{
     srch->TotalCycle++;

     InitSoln(currsoln,seed);
     srch->tbest = CpuTime();
     //printf("\n initial solution---------\n");
     //PrintSoln(currsoln);


     Search(srch,currsoln,seed);
     //PrintSoln(currsoln);
     if (currsoln->z>zbest)
     {
      tbest=srch->tbest;
     zbest=currsoln->z;
     bestsoln=*currsoln;
     srch->CycleBest=srch->TotalCycle;

     }
     *currsoln=tmpsoln;
}
*currsoln=bestsoln;
     srch->tbest=tbest;
}
```

main.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Slots.h"
#include "Choices.h"
#include "Modes.h"
#include "util.h"
#include <time.h>
#include <vector>
using namespace std;


float TT=0; /* Total Processing Time*/

void main( int argc, char *argv[] )
{
time_t curtime;                 /* stores calendar time for output */
double    srch_time,TimeToBest; /* Time markers */



ChoiceArray *ChoiceInstance;
ChoiceInstance=new ChoiceArray;     //a instance of Choice Class

Soln currsoln;// current, best solutions
Srch srch;
int Option;
int num=0,n;
long seed=123;
char * tmp;
char dfile[48]="";                  //data file name
char basename[48]="";               //base file name
curtime = time(NULL);

     printf( "Local Search Program Run ,     %s\n", ctime(&curtime) );

      printf("\t\t 1 - Generate the test problem \n");
     printf("\t\t 2 - Search & Solve the problem \n");
     printf("\t\t 0 - Exit \n");

     printf("\t\t Select Options: ");
     scanf("%d",&Option);
```

```
switch (Option)
{
case 1:

     if ( argc < 2 )
     {
          printf( "Input -> Base data    filename: ");
          scanf( "%s", basename);

      }
     else
     {
           strcpy( basename, argv[1] );
          //strcpy( dfile, argv[3] );
     }

    // printf(" how many instance you want to generate?\n");
     //scanf("%s",dfile);
    // num=atoi( dfile );
      num=5;
     ChoiceInstance->Tasks_i->SetBaseFileName(basename);
     ChoiceInstance->Tasks_i->InputBase();
      for(n=1;n<=num;n++)
     {
     sprintf( dfile,
"%02d%02d%01d%d.%2.0f",ChoiceInstance->Tasks_i->Base.Horizon,

     ChoiceInstance->Tasks_i->Base.NrOfTasks,ChoiceInstance->Tasks_i->Base.MaxM
ode,
          n,ChoiceInstance->Tasks_i->Base.R1RS*100);
      //ChoiceInstance->Tasks_i->Base.Seed=ChoiceInstance->Tasks_i->Base.Seed*2;
     ChoiceInstance->Tasks_i->EmptyTask();
     ChoiceInstance->Tasks_i->GeneTasks();
     ChoiceInstance->Tasks_i->WriteTasks(dfile);
     }
     break;

case 2:
   if ( argc < 2 ) {
      printf( "Input -> problem    filename: ");
      scanf( "%s", dfile);

   }
   else
   {
```

```
    strcpy( dfile, argv[1] );
  }

ChoiceInstance->Tasks_i->SetDataFileName(dfile);
ChoiceInstance->Tasks_i->ReadTasks();

ChoiceInstance->CreateSlots();
ChoiceInstance->GenChoices();
ChoiceInstance->Tasks_i->Print();
srch.insertmoves=0;
srch.swapiter=0;
srch.feasibleswap=0;
srch.iterbest=0;
srch.insertiter=0;
srch.TotalCycle=0;
srch.CycleBest=0;
srch_time = CpuTime();                /* mark start of execution */
ChoiceInstance->MultiStart(&srch,&currsoln,&seed,30);
TimeToBest= srch.tbest-srch_time;
srch_time = CpuTime() - srch_time;



// output the best solution
/* Output Search Statistics */

   printf( "\n======= Best Solution   Found ==========\n\n");
   printf( "\n At MultiStart Cycle %d of %d\n\n", srch.CycleBest,srch.TotalCycle);
   ChoiceInstance->PrintSoln(&currsoln);
   tmp = ClockTime( srch_time );
   printf(" Totel search time is %s \n\n",tmp);
   tmp = ClockTime( TimeToBest );
   printf(" Time to the best solution is   %s \n\n",tmp );


   printf(" Total Swaps to evaluate   is   %d \n\n",srch.swapiter);
   printf(" Total Feasible Swaps      is   %d \n\n",srch.feasibleswap);
   printf(" Total Swaps taken is   %d \n\n",srch.iterbest);

   printf(" Total Inserts to evaluate   is   %d \n\n",srch.insertiter);
   printf(" Total Insert moves taken is   %d   \n\n", srch.insertmoves);



printf("----------------------\n");
```

```
break;

case 3:
      exit(0);
}   //end switch

delete ChoiceInstance;
}
```

APPENDIX B

GNU LP-SOLVER SOUCE CODE

LP.c

```
#include <stdarg.h>

#include <stdio.h>
#include <stdlib.h>
#include "glpk.h"
#include <time.h>

#define Print_data 1

#define MPS    0
#define DEBUG 1
#define ArraySize 6
int MIP=0;

LPX *BldLP ( FILE *outfile );
double CpuTime(void);
char *ClockTime( float ss );
void press_ret(void);
void print_results(LPX *lp);

/*define globle variable*/
   int NTasks,Horizon;
   int iseed;    /* Starting Seed Value */
   int lowp,highp; /* Uniform[lowp,highp] for the processing times generation */
   int lowpu,highpu; /* Uniform[lowpu,highpu] for the parallel machine request*/
   int setupR2;/*setup resource compacity*/
   int TotalMode=0;    //total number of modes
   int NR1,NR2,NR3;    //resource number of each type
   int AR1[ArraySize],AR2[ArraySize],AR3[ArraySize];    //resource compacity of
R1,R2,R3

   struct modeinfo
   {
       int modeindex;
      int duration;
       int R1[ArraySize];
      int R2[ArraySize];
      int R3[ArraySize];

    } ;

   struct jobinfo
   {
```

```
        int tasknumber;
        int modenumber;
       struct modeinfo mode[ArraySize];
    } Tasks[1000];

int main(int argc, char *argv[])
{

   float   srch_time; /* Time markers */
   char * tmp;
   int i,m,r,ret;
   LPX *lp;

   FILE   *outfile;

   FILE *infile;                  /* Problem data file pointer */
   char probnam[20]="";           /* problem data set name */
   char dfile[48]="";            /* problem data file (probnam.dat) */
   char mpsfile[48]="";           /* MPS output file (MPS.dat) */
   char out_fname[48]="";

   /* read data from file */
   if ( argc < 2 ) {
      printf( "Input -> Problem filename: ");
      scanf( "%s", dfile );
   }
   else
      strcpy( dfile, argv[1] );

   if (argc ==3){
      if (strcmp(argv[2], "I")==0) MIP=1;    /* argv[2]=I MIP */
         else MIP=0;
   }
   if((infile=fopen(dfile,"r"))==NULL) {
      printf("%s open failed\n",dfile);
                        exit(-1);
            }
   else
      printf("%s opened\n\n",dfile);



   /* Now, read the problem */
   fscanf( infile, "%d   \n", &iseed );
   fscanf( infile, "%3d %3d\n", &NTasks,&Horizon);
```

```
    fscanf( infile, "%3d %3d %3d\n", &NR1,&NR2,&NR3);
    for(i=0;i<NR1;i++)
        fscanf( infile, "%3d",&AR1[i]);
    for(i=0;i<NR2;i++)
        fscanf( infile, "%3d",&AR2[i]);
    for(i=0;i<NR3;i++)
        fscanf( infile, "%3d",&AR3[i]);
    fscanf(infile,"\n");
  for (i=0; i<NTasks; i++ ) {              /* get job information*/

      fscanf( infile, "%3d %3d",&Tasks[i].tasknumber,&Tasks[i].modenumber);
     TotalMode=TotalMode+Tasks[i].modenumber;
     for(m=0;m<Tasks[i].modenumber;m++)
     {
        fscanf( infile, "%3d",&Tasks[i].mode[m].modeindex);
        fscanf( infile, "%3d",&Tasks[i].mode[m].duration);
       for(r=0;r<NR1;r++)
            fscanf(infile, "%3d",&Tasks[i].mode[m].R1[r]);
       for(r=0;r<NR2;r++)
            fscanf(infile, "%3d",&Tasks[i].mode[m].R2[r]);
       for(r=0;r<NR3;r++)
            fscanf(infile, "%3d",&Tasks[i].mode[m].R3[r]);
     }
     fscanf(infile,"\n");
    }

    /*output to screen for test*/
    if (Print_data==1)
    {
    printf("%d    \n", iseed );
    printf( "%3d %3d\n", NTasks,Horizon);
    printf("%3d %3d %3d\n", NR1,NR2,NR3);
    for(i=0;i<NR1;i++)
        printf( "%3d",AR1[i]);
    for(i=0;i<NR2;i++)
        printf("%3d",AR2[i]);
    for(i=0;i<NR3;i++)
        printf("%3d",AR3[i]);
    printf("\n");

  for (i=0; i<NTasks; i++ ) {              /* get job information*/

      printf("%3d %3d",Tasks[i].tasknumber,Tasks[i].modenumber);
     for(m=0;m<Tasks[i].modenumber;m++)
     {
```

```
        printf("\t %3d",Tasks[i].mode[m].modeindex);
        printf( "%3d",Tasks[i].mode[m].duration);
        for(r=0;r<NR1;r++)
            printf("%3d",Tasks[i].mode[m].R1[r]);
        for(r=0;r<NR2;r++)
            printf("%3d",Tasks[i].mode[m].R2[r]);
        for(r=0;r<NR3;r++)
            printf("%3d",Tasks[i].mode[m].R3[r]);
        printf("\n");
      }

  }
}

  lp = BldLP ( outfile );


  if(MPS) {
     sprintf( mpsfile, "%s.mps", dfile );
     i=lpx_write_mps(lp,mpsfile);}
  else {
     sprintf( mpsfile, "%s.lp", dfile );
     i=lpx_write_lpt(lp,mpsfile);}
  if(i!=0)
    {
     printf("%s write    failed\n",mpsfile);
                    exit(-1);
     }
  else
     printf("%s write    success\n\n",mpsfile);


  /*    write_LP(lp,outfile);*/
  press_ret();



  lpx_set_int_parm(lp, LPX_K_MSGLEV, 3); /*output all information*/
  lpx_set_int_parm(lp, LPX_K_BRANCH, 0); /*branch on first variable*/
  lpx_set_int_parm(lp, LPX_K_BTRACK, 0); /*depth first search*/
  lpx_set_int_parm(lp, LPX_K_OUTFRQ, 300);/*300 "interval    to output */
  // lpx_set_int_parm(lp, LPX_K_ITLIM, 1000);/*set the iteration limit*/
  //lpx_set_real_parm(lp, LPX_K_TMLIM,60); /*set to 1 minutes*/

lpx_scale_prob(lp); /* scale the problem (optional) */
```

```
/* construct advanced initial basis (optional) */
   lpx_adv_basis(lp);

   lpx_set_real_parm(lp, LPX_K_TOLPIV, 1e-8);

   lpx_warm_up(lp);
   if (DEBUG){
   i=lpx_get_status(lp);
   switch (i)
      {
      case LPX_INFEAS:
         printf("\n solution is infeasible\n");
         break;
      case LPX_NOFEAS:
         printf("\n problem has no feasible solution\n");
         break;
      }
 }
   srch_time = CpuTime(); /* mark start of execution */
   /* find optimal solution of LP relaxation */

    ret=lpx_simplex(lp);

   /* make sure that the simplex method has successfully solved lp */
    insist(ret == LPX_E_OK);

   /* make sure that optimal solution exists */
    insist(lpx_get_status(lp) == LPX_OPT);

   if(MIP)
   {
      lpx_integer(lp);
    }
   srch_time = CpuTime() - srch_time;
   tmp = ClockTime( srch_time );

   print_results(lp);
   printf(" Totel search time is %s \n\n",tmp);

   sprintf( out_fname, "%s.out", dfile );
   if(MIP) lpx_print_mip(lp, out_fname);
      else lpx_print_sol(lp, out_fname);

   lpx_delete_prob(lp);
```

```
   return(0);
}

void press_ret(void)
{
   printf("[return]");
   getchar();
}

/* ---------------------------- BldLP
   Given a problem, builds an LpSolve format LP (used by LPKit)
   created, 11/17/02
*/
LPX *BldLP ( FILE *outfile )
{
   int *rn,*cn;
   double *aa;
   int colindex,i,m,t,r;
   LPX *lp;
   int index=0;
   int ii,tt;
   double *Row;
   char   xname[31+1],yname[31+1];
   lp = lpx_create_prob(); /* create mip problem instance */
   if(MIP)
      {
   lpx_set_class(lp,LPX_MIP);
      }
   lpx_set_prob_name(lp, "parallel");

   lpx_add_cols(lp,TotalMode*Horizon);
   lpx_add_rows(lp,NTasks+(NR1+NR2)*Horizon+NR3);
   lpx_set_obj_dir(lp, LPX_MAX);          /*Maximize throughput    */

   colindex=0;    /*set colume name */
   for(i=0;i<NTasks;i++)
        for(m=0;m<Tasks[i].modenumber;m++)
             for(t=0;t<Horizon;t++)
      {
         colindex++;
         sprintf( xname, "X%d_%d_%d", i+1,m+1,t+1 );
         lpx_set_col_name(lp,colindex,xname);

         lpx_set_col_bnds(lp, colindex, LPX_DB, 0.0, 1.0);
```

```
        lpx_set_col_coef(lp, colindex, 1.0);    /*set objective coefficent */

    }

        rn = ucalloc(1 + (TotalMode+1)*
(Horizon+1)*(NTasks+(NR1+NR2)*Horizon+NR3+1), sizeof(int));
        cn = ucalloc(1 + (TotalMode+1)*
(Horizon+1)*(NTasks+(NR1+NR2)*Horizon+NR3+1), sizeof(int));
        aa = ucalloc(1 + (TotalMode+1)*
(Horizon+1)*(NTasks+(NR1+NR2)*Horizon+NR3+1), sizeof(double));

  ii=0;
  for ( i=0; i<NTasks; i++ ) /* name the variables */
  {
      for(m=0;m<Tasks[i].modenumber;m++)
           for(t=0;t<Horizon;t++)
   {
        ii++;
        index++;
         rn[index]=i+1;
         cn[index]=ii;
         aa[index]=1;
   }
    lpx_set_row_bnds(lp, i+1, LPX_UP, 0.0, 1.0);
    sprintf( yname, "j%d", i+1 );
    lpx_set_row_name(lp, i+1, yname );    /* primal variable name */

  }


 for(r=0;r<NR1;r++)
 {
    for(t=0;t<Horizon;t++)
  {
      ii=0;
    for ( i=0; i<NTasks; i++ )
        for(m=0;m<Tasks[i].modenumber;m++)
        {
            for(tt=0;tt<Horizon;tt++)
        {
            ii++;
            if(tt>=t-Tasks[i].mode[m].duration+1 &&tt<=t
&&Tasks[i].mode[m].R1[r]!=0)
            {
                index++;
```

```
                rn[index]=NTasks+r*Horizon+t+1;
                cn[index]=ii;
                  aa[index]=Tasks[i].mode[m].R1[r];
              }
            }
      }

      sprintf( yname, "R1_%d_t%d",r,t );
      lpx_set_row_name( lp, NTasks+r*Horizon+t+1, yname );    /* primal variable name
*/
      lpx_set_row_bnds( lp, NTasks+r*Horizon+t+1,LPX_UP,0.0,AR1[r]);

     } //end for t
  } //end for r
      /*printf("%d",setupR2); for debug purpose*/

   /* add setup resouce constraints */

   for(r=0;r<NR2;r++)
  {
      for(t=0;t<Horizon;t++)
   {
        ii=0;
      for ( i=0; i<NTasks; i++ )
          for(m=0;m<Tasks[i].modenumber;m++)
          {
              for(tt=0;tt<Horizon;tt++)
          {
              ii++;
              if(tt==t && Tasks[i].mode[m].R2[r]!=0)
              {
                  index++;
                  rn[index]=NTasks+NR1*Horizon+r*Horizon+t+1;
                  cn[index]=ii;
                    aa[index]=Tasks[i].mode[m].R2[r];
              }
            }
      }

      sprintf( yname, "R2_%d_t%d",r,t );
      lpx_set_row_name( lp, NTasks+NR1*Horizon+r*Horizon+t+1, yname );    /* primal
variable name */
      lpx_set_row_bnds( lp,
NTasks+NR1*Horizon+r*Horizon+t+1,LPX_UP,0.0,AR2[r]);
```

```
    } //end for t
  } //end for r

    for(r=0;r<NR3;r++)
{
      ii=0;
    for ( i=0; i<NTasks; i++ )
        for(m=0;m<Tasks[i].modenumber;m++)
        {
            for(tt=0;tt<Horizon;tt++)
            {
            ii++;
              if(Tasks[i].mode[m].R3[r]!=0)
              {
                index++;
                rn[index]=NTasks+(NR1+NR2)*Horizon+r+1;
                cn[index]=ii;
                  aa[index]=Tasks[i].mode[m].R3[r];
              }
            }
        }

    sprintf( yname, "R2_%d_t%d",r,t );
    lpx_set_row_name( lp, NTasks+(NR1+NR2)*Horizon+r+1, yname );   /* primal
variable name */
    lpx_set_row_bnds( lp, NTasks+(NR1+NR2)*Horizon+r+1,LPX_UP,0.0,AR3[r]);

 } //end for r

  //index=1 + (TotalMode+1)*(Horizon+1)*(NTasks+(NR1+NR2)*Horizon+NR3+1);
  //for(i=1;i<index+1;i++)
    //   printf("%d   -> %d_%d=%f\n",i, rn[i],cn[i],aa[i]);

  //exit(0);

  lpx_load_mat3(lp,index , rn, cn, aa);
  ufree(rn);
  ufree(cn);
  ufree(aa);
  if (MIP)
    for(i=1;i<=TotalMode*Horizon;i++)
    lpx_set_col_kind(lp,i,LPX_IV);

  return lp;
```

```
} /* end BldLP() */

void print_results(LPX *lp)
{
  int i,m,t,tt,jj,colindex,IntTotal=0,TotalVars;
  double Z,x;

  if(MIP)
    Z=lpx_get_mip_obj(lp);
  else
    Z=lpx_get_obj_val(lp);

  /* fprintf(stderr, "The total number of iterations(B&B) is %d \n",lp->total_iter);*/
  fprintf(stderr, "The Objective value is %g \n",Z);

  /* i=0;
   for(jj=1;jj<=NTasks;jj++)
    for(tt=1;tt<=Horizon;tt++)
    {
      i++;
      if(MIP){
      if( lpx_get_mip_col(lp,i)!=0)
       printf("X%d_%d=%g\n", jj,tt,lpx_get_mip_col(lp,i));
      }
      else
      {
        lpx_get_col_info(lp,i,NULL,&x,NULL);
        if (x!=0) printf("X%d_%d=%f\n", jj,tt,x);
       }

    }
*/
    //-----------------------------
    printf("LP solver results\n");
  colindex=0;   /*set colume name */
  for(i=0;i<NTasks;i++)
      for(m=0;m<Tasks[i].modenumber;m++)
          for(t=0;t<Horizon;t++)
    {
      if(x==0 || x==1)   IntTotal++;
       colindex++;
    if(MIP)
    {
       if( lpx_get_mip_col(lp,colindex)!=0)
        printf("X%d_%d_%d=%f\n", i,m,t,lpx_get_mip_col(lp,colindex));
```

```
        }
      else
      {
         lpx_get_col_info(lp,colindex,NULL,&x,NULL);
         if (x!=0) printf("X%d_%d_%d=%f\n", i,m,t,x);
      }

       }
       TotalVars=lpx_get_num_cols(lp);
      printf("Total Constraints is %d \n",lpx_get_num_rows(lp));
      printf("Total Variables is %d \n",TotalVars);
      printf("Total int Variables results is %d \n",IntTotal);
      printf("    is %9.2f %\n",(float)IntTotal/TotalVars*100);

}   /*debug_print_solution */

double CpuTime(void)
{

 return (double) clock() / (double) CLOCKS_PER_SEC; /* may need this CLK_TCK; */


} /* end CpuTime() */

char *ClockTime( float ss )
{
 int hh, mm;
 static char time[15] = "00:00:00.00";

   hh = ss / 3600;
   ss = fmod( ss, 3600.0 );
   mm = ss / 60;
   ss = fmod( ss, 60.0);
   sprintf( time, "%02d:%02d:%05.2f", hh, mm, ss );

   return time;

} /* end ClockTime() */
```

APPENDIX C

RESULTS OF COMPUTATIONAL EXPERIENCES

GNU LP-solver output for T=10,NR1= NR2=1,ar1= ar2=1, d(i)=[1,10]

```
/* Integer results*/
101011.1 opened

123          //seed
 10   10     // horizon , tasks number
  1    1     0   // NR1,NR2,NR3
  6   2       //Ar1,Ar2
  1    1      1   6   1   1   //Task index; mode number;mode index;duration;ar1,ar2
  2    1      1 10   1   1
  3    1      1   6   1   1
  4    1      1   3   1   1
  5    1      1   1   1   1
  6    1      1   3   1   1
  7    1      1   6   1   1
  8    1      1 10   1   1
  9    1      1   2   1   1
 10    1      1   7   1   1
lpx_write_lpt: writing problem data to `101011.1.lp'...
101011.1.lp write    success

[return]gm_scal: max / min = 1.000e+000
gm_scal: max / min = 1.000e+000
lpx_adv_basis: size of triangular part = 30
*       0:    objval =   0.000000000e+000    infeas =   0.000000000e+000 (0)
*      26:    objval =   1.000000000e+001    infeas =   0.000000000e+000 (0)
OPTIMAL SOLUTION FOUND
LP solver results
X0_0_0=1.000000
X1_0_0=1.000000
X2_0_1=1.000000
X3_0_1=1.000000
X4_0_2=1.000000
X5_0_2=1.000000
X6_0_3=1.000000
X7_0_4=1.000000
X8_0_5=1.000000
X9_0_6=1.000000
Total Constraints is 30
Total Variables is 100
Total int Variables results is 100
Totel search time is 00:00:32.00
lpx_print_sol: writing LP problem solution to `101011.1.out'...
```

```
/* Noninteger results*/

101014.1 opened

989958143
 10   10
  1    1    0
  6   2
  1    1        1 10   1   1
  2    1        1   6   1   1
  3    1        1 10   1   1
  4    1        1   4   1   1
  5    1        1   6   1   1
  6    1        1   8   1   1
  7    1        1   6   1   1
  8    1        1   1   1   1
  9    1        1   4   1   1
 10    1        1   8   1   1
lpx_write_lpt: writing problem data to `101014.1.lp'...
101014.1.lp write    success

[return]gm_scal: max / min = 1.000e+000
gm_scal: max / min = 1.000e+000
lpx_adv_basis: size of triangular part = 30
*       0:     objval =   0.000000000e+000     infeas =   0.000000000e+000 (0)
*      39:     objval =   1.000000000e+001     infeas =   0.000000000e+000 (0)
OPTIMAL SOLUTION FOUND
LP solver results
X0_0_0=0.875000
X0_0_6=0.125000
X1_0_0=1.000000
X2_0_1=1.000000
X3_0_1=1.000000
X4_0_2=0.875000
X4_0_6=0.125000
X5_0_0=0.125000
X5_0_2=0.875000
X6_0_5=1.000000
X7_0_2=0.250000
X7_0_3=0.250000
X7_0_4=0.250000
X7_0_5=0.250000
X8_0_6=1.000000
X9_0_8=1.000000
Total Constraints is 30
```

Total Variables is 100
Total int Variables results is 89 %

Totel search time is 00:00:32.00

lpx_print_sol: writing LP problem solution to `101014.1.out'...

EXAMPLE OUTPUT FOR LOCAL SEARCH ALGORITHM

./mode3/101031.10 opened

Test Data file name is ./mode3/101031.10
Seed = 123
Tasks Number = 10 Horizon =10
The number of R1,R2,R3 resources = 3 3 3

Task	Mode	Dur	13	13	14	17	15	11	38	38	36	
1	1	6	4	6	5	6	9	7	9	1	5	
1	2	6	9	9	7	1	3	4	8	10	9	
1	3	3	6	4	10	6	8	4	7	5	3	
2	1	10	7	2	3	7	7	3	7	7	6	
2	2	5	7	7	4	5	5	4	5	5	3	
2	3	9	7	4	10	9	8	1	2	9	8	
3	1	10	4	4	9	4	1	6	8	8	10	
3	2	6	1	8	7	5	10	6	4	8	1	
3	3	9	6	10	9	1	2	4	9	7	8	
4	1	7	10	8	3	10	10	6	5	2	1	
4	2	9	3	3	5	4	8	4	2	2	8	
4	3	8	1	5	2	8	7	8	1	6	3	
5	1	9	8	3	10	10	5	2	6	6	1	
5	2	3	8	8	1	4	8	3	4	7	5	
5	3	4	3	9	9	6	10	4	1	8	2	
6	1	1	3	6	8	5	1	4	10	1	9	
6	2	4	9	10	2	1	5	1	6	5	6	
6	3	2	5	9	4	8	10	3	8	7	8	
7	1	10	2	2	6	2	9	3	8	7	9	
7	2	6	8	7	9	8	3	7	5	2	7	
7	3	7	9	1	5	4	3	3	3	9	7	
8	1	1	7	8	4	5	6	7	9	5	4	
8	2	1	8	6	4	5	3	1	8	1	7	
8	3	10	4	9	10	1	5	2	6	5	9	
9	1	8	5	3	1	9	6	7	4	7	3	
9	2	3	8	7	7	10	6	4	2	3	8	
9	3	7	10	3	3	10	1	9	1	5	6	
10	1	8	4	10	5	6	3	9	2	7	10	
10	2	4	7	7	9	3	3	10	6	5	1	
10	3	1	10	6	7	9	5	4	8	8	3	

======= Best Solution Found ==========

At MultiStart Cycle 52 of 1601

Solution results
Objective Function Value is Max Z= 6.00
Constraints vialations are 0

Task	Start	Duration	Resource request units								
1	10	6	4	6	5	6	9	7	9	1	5
6	10	1	3	6	8	5	1	4	10	1	9
9	6	3	8	7	7	10	6	4	2	3	8
8	2	1	8	6	4	5	3	1	8	1	7
4	1	8	1	5	2	8	7	8	1	6	3
10	4	1	10	6	7	9	5	4	8	8	3

Totel search time is 00:00:30.01

Time to the best solution is 00:00:01.01

Total Swaps to evaluate is 2807220

Total Feasible Swaps is 183029

Total Swaps taken is 10

Total Inserts to evaluate is 559076

Total Insert moves taken is 638