A SURVEY OF ALGORITHMS FOR SCHEDULING NON-INTERRUPTIBLE TASKS

By

SHENGLI CAO

Bachelor of Civil Engineering

Taiyuan Institute of Technology

Shanxi, China

1982

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the degree of
MASTER OF SCIENCE
July, 2000

A SURVEY OF ALGORITHMS FOR SCHEDULING

NON-INTERRUPTIBLE TASKS

Thesis Approved:

J Chandler

Thesis Adviser

[illegible]

[illegible]

[illegible]

Dean of the Graduate College

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my advisor, Dr. John P. Chandler, for his continuous guidance, concern, patience, and support throughout my graduate program. I shall always be thankful for his inspiration and advice on my study of computer science, his encouragement and help for me to apply computer science theory to real industrial practice.

I would also like to express my gratitude to my thesis committee members, Dr. Blayne Mayfield, Dr. Jacques LaFrance and Dr. H.K. Dai for their helpful suggestions and assisting with my research.

I wish to thank all the faculties and staff members of Computer Science Department who are always kind and helpful to me. I extend my thanks to Oklahoma State University for giving me the opportunity to accomplish my graduate program.

Finally, I would express my thanks to my parents for their lifelong love and support. I also express my deepest gratitude to my daughter, Lu Cao, always gave me encouragement throughout my study.

TABLE OF CONTENT

Chapter . Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Scheduling is an effective allocation of a set of machines over time to a set of jobs [Silberschatz and Galvin 1998].

Suppose that we have to perform a number of jobs, each of which consists of a given sequence of operations, by using a number of machines. We want to find a processing order on each machine so that the corresponding cost is minimized. This is scheduling.

It is true that scheduling originally arose in an industrial production context. However, various other interpretations are possible: jobs and machines can stand for patients and hospital equipment, classes and teachers, ships and dockyards, dinners and cooks, programs and computers, or cities and salesmen, and so on. Each of these situations fits into the framework sketched above and thus falls within the scope of machine scheduling theory and algorithms.

In job processing, some problems allow jobs to be interrupted or preempted, others do not. If interruption or preemption is not allowed, then the machine must process the job continuously until it is finished, once a job has begun on a given machine. We call such problem "Non-interruptible Scheduling." This thesis focuses on algorithms for non-interruptible scheduling problems. A survey of the algorithms for non-interruptible scheduling problems is presented in this thesis. Some simple examples of the execution of these algorithms are also provided to illustrate the algorithms. To show a comprehensive application of these algorithms, I also present an implementation of a real work force dispatching project developed for the customer service and maintenance division of an energy company. Comparisons have been made of the implementation of different algorithms to see the performance of these algorithms and real world requirements for an approximately optimal result.

CHAPTER 2

SURVEY OF PROBLEMS AND ALGORITHMS

The study of scheduling is motivated by problems that arise in production planning, in project management, in military movement, in computer control, and so on. In general, these problems are from a situation in which scarce resources have to be allocated to jobs over time. Due to the demand for optimal scheduling by fast developing industries, scheduling theory and applications have become the subject of extensive study and research since early the 1950's. Much of the literature has been focused on algorithms for solving all kinds of problems since then. This paper, as mentioned in the preceding section, will concentrate on the basic problems and algorithms in the area of deterministic non-interruptible machine scheduling. Attention will also be given to combinatorial optimization by using genetic algorithms and special methods for traveling salesman problems.

The survey is organized into five sections. Section 1 classifies scheduling problems and notations. Section 2 presents the algorithms for single machine scheduling. Section 3 contains algorithms for open shop, flow-shop and job-shop problems. Section 4 provides algorithms for parallel machine problems. Section 5 discusses some problems in combinatorial optimization.

2.1 Problem Classification And Notation

Scheduling is an effective allocation of a set of machines over time to a set of jobs. Suppose that m machines M_i (i = 1, 2, ..., m) have to process n jobs J_j (j = 1, 2, ..., n). A non-interruptible scheduling is, therefore, an allocation of a time interval on one machine for each job. Non-interruptible scheduling is feasible if no two time intervals on the same machine overlap, and if the schedule meets a number of specific requirements concerning machine environments and the job characteristics. A non-interruptible scheduling is optimal if it minimizes a given optimality criterion. So, the machine environments, the job characteristics and the optimality criteria together define a problem type. Some literature specifies scheduling problems in terms of a three-field classification, while others use four or more fields that provide extra sections to define the machine and processing environments.

2.1.1 Three-Field Classification

The three-field classification is denoted by $\alpha/\beta/\gamma$. [Conway, Maxwell, and Miller 1967], [Lawler, Lenstra, and Rinnooy Kan 1982], [Herrmann, Lee and Snowdon 1993]. The α field describes the machine environment and contains a single entry, the β field provides details of processing characteristics and constraints and may contain no entry, a single entry, or multiple entries, and the γ field specifies the optimality criterion of interest and usually contains a single entry. This classification is introduced as:

a) Job Data

A job is generally specified by the following data:

The number of jobs denoted by *n* (n is assumed to be finite).

Processing time (P_{ij}) ---- the processing time of job *j* on machine *i*. *i* is omitted if job j does not depend on the machine or if job *j* is only to be processed on one given machine. The process time is usually assumed to be known fairly precisely. But this is not always true. For example, transmitting files over modems, the transmission rate depends on the degree of congestion in the network and processing time can vary considerably. Similarly convalescence time for hospital patients can be unpredictable, etc.

Release date (r_j) ---- the time on which a job becomes available for processing, also referred as the ready date.

Due date (d_j) ---- the date a job is promised to the customer.

b) Machine Environment

The machine environment indicates the number of machines and describes the configuration of the processing environment relative to the machines. The following specifications are generally used:

Single machine (1) ---- a single machine environment; $P_{1j} = P_j$.

Identical machine in parallel (*Pm*) ---- m identical machines in parallel,

$P_{ij} = P_j$ for all M_i;

Machines in parallel with different speeds (*Qm*) ---- m machines in parallel with different speeds. Machine speeds are independent of jobs.

Unrelated machines in parallel (*Rm*) ---- m different machines in parallel.

Machine speeds depend on the jobs processed.

Flow shop (*F*) ---- m machines in series, jobs possess multiple operations. The ordering is the same for each job. For example, in an assembly shop, a number of operations have to be done on every job. Often, these operations have to be done on all jobs in the same order, which implies that the jobs have to follow the same route. The machines are assumed to be set up in series and the environment is referred as a flow-shop.

Job shop (*J*) ---- m machines in series, jobs possess multiple operations.

The ordering is not required for each job.

Open shop (*O*) ---- m machines, each job may be processed more than once on each of the m machines but no ordering on the machines is imposed.

c) Job Characteristics

Generally, the job characteristics specify whether or not a job can be interrupted during processing, whether or not precedence ordering is imposed on jobs, whether or not job-dependent release times are given, and finally, specifications regarding job duration times, e.g. all jobs possess unit duration times.

Preemption (*pmtn*) ---- Jobs are interruptible during processing. All jobs in this thesis are non-interruptible (i.e. there is no preemption).

Precedence constraints (*prec*) ---- Precedence requires that one or more jobs may have to be completed before another job is allowed to start its processing. If each job has at most one predecessor and one successor, the constraints are referred to as *chains*. If each job has at most one successor, the constraints are referred to as *intree*. If each job at most has one predecessor, the constraints are referred to as *outtree*. If no precedence constraints appear in the β field, the jobs are not subjected to precedence constraints.

Job-dependent release time (r_j) ---- release dates that may differ per job.

Unit duration time (t_i)---- all jobs possess unit duration times $t_i = 1$.

d) Optimality Criteria

The field γ is used for specifying measures of performance and, as a consequence, is generally self-evident.

For example, Let C_j denote completion time, d_j due date, L_j lateness, *and Tj* tardiness

$L_j = C_j - d_j$

$T_j = \max\{0, C_j - d_j\}$

(T_j is kind of lateness, but it is always a positive number, L_j can either be positive or a negative number which represents the job completed ahead of due day).

The optimality criteria commonly chosen involve the minimization of maximum completion time C_{max} (sometimes called the schedule length or makespan) and the minimization of L_{max} and T_{max}.

2.1.2 Four-Field Classification

The four-field classification is denoted by $n/m/A/B$. [Conway, Maxwell, and Miller 1967], [Lenstra 1977], [Rinnooy Kan 1976], [Graham et al. 1979] and [French 1982].

Where

n is the number of jobs.

m the number of machines.

A describes the flow pattern or discipline within the machine shop.

When $m = 1$, A is left blank. A may be

F ---- the flow-shop case

P ---- the permutation flow-shop case

G ---- the general job-shop case

B provides optimality criteria, the same as above.

This paper will use three-field classification and four-field classification alternatively. Notation will be given if a specific classification is used in the section.

2.2 Algorithms For Single Machine Scheduling

(In this section, problems are described by a three-field classification $\alpha/\beta/\gamma$.)

Single machine models often display properties that do not hold for either machines in parallel or machines in series. A single machine environment provides a basis for heuristics for more complicated machine environments. In practice, scheduling problems in more complicated machine environments are often decomposed into sub-problems that deal with single machines.

2.2.1 Permutation Schedules

Assume we have a job list { $J_1, J_2, \ldots, J_n$ } and a machine M. If jobs are scheduled without designated idle time, the schedule is perfect. Otherwise we must have jobs permuted with minimum designated idle time. That is, the machine starts processing at time equals zero and continues without or with minimum rest until the time equals total processing time or with a minimum amount of extra time.

In scheduling, we write $J_{j(k)}$ for job J_j that is scheduled at the *k*th position in the processing sequence. Thus J_j is simply a generic job drawn from the job list $\{J_1, J_2, \ldots, J_n\}$ and $J_{j(k)}$ is the job that the processing sequence selects as the *k*th to be processed, $k = 1$, 2, …, n. Permutation usually follows the FIFO discipline. Calculation of the total cost is straightforward, so no detailed description is necessary here. See Fig. 1.

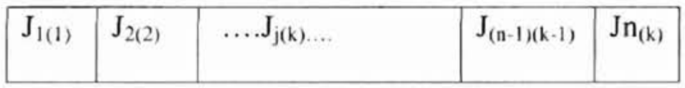

Fig. 1 Gantt Diagram Of Permutation Scheduling

2.2.2 Shortest Processing Time Scheduling (SPT)

For a particular job, the average job flow time $\bar{F}$ is calculated as follow:

$$\bar{F} = \frac{1}{n}\sum_{j=1}^{n} F_j = \frac{1}{n}\sum_{j=1}^{n}(W_j + P_j)$$

$$= \frac{1}{n}\sum_{j=1}^{n} W_{j(k)} + \frac{1}{n}\sum_{j=1}^{n} P_{j(k)}$$

where F ---- job flow time

W---- job waiting time

P ---- job processing time

Hence $\sum_{k=1}^{n} P_{j(k)} = \sum_{j=1}^{n} P_j$ is a constant for all sequences. Hence to minimize $\bar{F}$ we must minimize $\sum_{k=1}^{n} W_{j(k)}$.

Therefore, for one machine and n jobs, minimizing the average flow time denoted by $(1/n/\bar{F})$, the mean flow time is minimized by sequencing $P_{j(1)} \leq P_{j(2)} \leq P_{j(3)} \leq \ldots\ldots \leq P_{j(n)}$.

Hence $P_{j(k)}$ is the processing time of the job that is processed *k*th.

Thus, a queue scheduled with the shortest processing time first will solve (one machine and n job) problems in minimizing the mean flow time, mean completion time, mean waiting time, mean lateness, mean number of unfinished jobs, and mean number of jobs waiting between machines, [Conway, Maxwell and Miller, 1967], [Rinnooy Kan 1976].

$P_{j(1)}$	$P_{j(2)}$	....	$P_{j(k-1)}$	$P_{j(k)}$

Fig.2 Gantt Diagram Of Schedules SPT

Example: One machine, six jobs, minimum mean flow time problem $(1/6/\bar{F})$

Job	1	2	3	4	5	6
Processing Time	6	5	4	3	2	1

The optimal SPT schedule for jobs is (6, 5, 4, 3, 2, 1). That is, the shortest job is scheduled first.

The flow time of each job is:

$F_{j(1)} = 1$

$F_{j(2)} = 1 + 2$

$F_{j(3)} = 1 + 2 + 3$

$F_{j(4)} = 1 + 2 + 3 + 4$

$F_{j(5)} = 1 + 2 + 3 + 4 + 5$

$F_{j(6)} = 1 + 2 + 3 + 4 + 5 + 6$

Mean flow time of the schedule is:

$$\frac{1}{6}\sum_{k=1}^{6} F_{j(k)} = \frac{1}{6}(6x1 + 5x2 + 4x3 + 3x4 + 2x5 + 6) = 9.3333$$

Clearly the schedule is optimal.

2.2.3 Earliest Due Date Scheduling (EDD) [Jackson 1955]

An initial approach to scheduling is, perhaps, to sequence jobs in the order in which they are required. In other words, sequence the jobs such that the first processed has the earliest due date, the second processed has the next earliest due date, and so on. For one machine, scheduling jobs by the earliest completion time and never delaying jobs results in processing the maximum number of jobs [Pinedo 1995], but skipping any jobs is usually not an option in real life.

Thus, for one machine and n jobs, minimizing maximum lateness denoted by ($1/n/L_{max}$), sequencing minimizes the maximum lateness

$$d_{j(1)} \leq d_{j(2)} \leq d_{j(3)} \leq \ldots \leq d_{j(n)}$$

where $d_{j(k)}$ is the due date of the job j that is processed kth.

$d_{j(1)}$	$d_{j(2)}$	....	$d_{j(k-1)}$	$d_{j(k)}$

Fig.3 Gantt Diagram Of Schedules EDD

Example: One machine, six jobs, minimizing the maximum tardiness problem ($1/6/T_{\max}$)

Job	1	2	3	4	5	6
Due Date	7	3	8	12	9	3
Processing Time	1	1	2	4	1	3

The optimal EDD schedule for jobs is (6, 2, 1, 3, 5, 4). From the table below, we find the optimal $T_{\max} = 1$.

Job	Completion Time	Lateness	Tardiness
$J_{j(k)}$	$C_{j(k)} = \sum_{l=1}^{k} P_{j(l)}$	$L_{j(k)} = C_{j(k)} - d_{j(k)}$	$T_{j(k)} = \max(0, C_{j(k)} - d_{j(k)})$
6	3	0	0
2	4	1	1
1	5	-2	0
3	7	-1	0
5	8	-1	0
4	12	0	0

2.2.4 Moore's Algorithm [Moore 1968], [Sturm 1970], [Kise et al. 1978]

In the real world, if a job is behind its due date by a few seconds or a few minutes, the entire production might be upset. Thus, we need to minimize the number of tardy jobs. An algorithm for solving this problem is due to Moore, but in a form suggested by Hodgson [Moore 1968].

Algorithm (Moore and Hodgson)

Step1: Sequence the jobs according to the EDD rule to find the current sequence

$(J_{j(1)}, J_{j(2)}, \ldots J_{j(n)})$ such that

For $k=1,2,\ldots,n-1$.

$d_{j(k)} \leq d_{j(k+1)}$

Step 2: Find the first tardy job, say $J_{j(l)}$, in the current sequence. If no such job is found, go to step 4.

Step 3: Find the job in the sequence $(J_{j(1)}, J_{j(2)}, \ldots J_{j(l)})$ with the largest processing time and reject this from the current sequence. Return to step 2 with a current sequence one shorter than before.

Step 4: Form an optimal schedule by taking the current sequence and appending to it the rejected jobs, which may be sequenced in any order.

Note: The rejected jobs will be tardy and these will be the only tardy jobs.

Example: One machine six jobs, minimizing the maximum number of tardy jobs.

Job	1	2	3	4	5	6
Due Date	15	6	9	23	20	30
Processing Time	10	3	4	8	10	6

1) Use the EDD sequence to compute the completion times until a tardy job is found (Steps 1 and 2):

New Sequence	2	3	1	5	4	6
Due Date	6	9	15	20	23	30
Processing Time	3	4	10	10	8	6
Completion Time	3	7	17			

2) We find job 1 to be the first tardy job in the sequence and of the sub-sequence (2, 3, 1) it has the largest processing time. So reject it (Step 3). Return to Step 2 with the new sequence:

New Sequence	2	3	5	4	6	Rejected jobs
Due Date	6	9	20	23	30	1
Processing Time	3	4	10	8	6	
Completion Time	3	7	17	25		

3) We find job 4 to be the first tardy job in the sequence and of the sub-sequence (2, 3, 5, 4), job 5 has the largest processing time. So reject it (Step 3). Return to Step 2 with the new sequence. No tardy jobs are now found:

New Sequence	2	3	4	6	Rejected jobs
Due Date	6	9	23	30	1, 5
Processing Time	3	4	8	6	
Completion Time	3	7	15	21	

4) Hence, we move on to Step 4 and form the optimal sequence (2,3,4,6,1,5).

2.2.5 Lawler's Algorithm [Lawler 1973]

This algorithm deals with general precedence constraints. Here we shall simply be constrained to process certain jobs before, but not necessarily immediately before, certain others. Lawler's algorithm minimizes the maximum cost of processing a job, where this cost has a general form $\gamma_j(C_j)$ for J_j and is taken to non-decreasing in the completion time C_j .Thus, the algorithm minimizes

$$\max_{j=1}^{n}\{\gamma_j(C_j)\}$$

Consider one-machine and n jobs with precedence constraints, minimizing the maximum cost problem denoted by $(1/n/\max_{j=1}^{n}\{\gamma_j(C_j)\})$. Let V denote the subset of jobs, which may be performed last (i.e. those jobs which are not required to precede any other jobs). Note that the final job in the schedule must be completed at $\tau=\sum_{j=1}^{n}P_j$. Let J_k be a job in V such that

$$\gamma_k(\tau)=\min_{J_j in V}\{\gamma_j(\tau)\},$$

(i.e. of all the jobs that may be performed last, J_k incurs the least cost). Then there is an optimal schedule in which J_k is scheduled last.

Example: One-machine and six jobs, minimizing maximum lateness problem with the precedence constraints in Fig. 4.

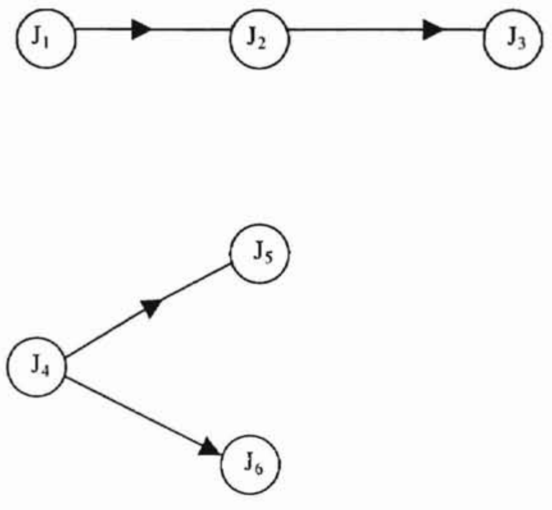

Fig. 4 Precedence Constraints

Job	J_1	J_2	J_3	J_4	J_5	J_6
Processing Time	2	3	4	3	2	1
Due Date	3	6	9	7	11	7

1) Find the job to be processed sixth: $\tau = 2 + 3 + 4 + 3 + 2 + 1 = 15$.

Jobs J_3, J_5, J_6 can be processed last, i.e. $V=\{J_3, J_5, J_6\}$. So the minimum lateness over V = min {(15-9), (15-11), (15-7)}, which occurs for J_5. Hence J_5 is scheduled sixth.

2) Find the job to be processed fifth:

Delete J_5 from our list and note that the completion time of the first five jobs $\tau = 15 - 2 = 13$. J_3, or J_6 can be processed last now; i.e. $V=\{J_3, J_6\}$. So the minimum lateness over V = min {(13-9), (13-7)}, occurs for J_3. So J_3 is scheduled fifth.

3) Find the job to be processed fourth:

J_3 and J_5 have been deleted from our list. J_2 becomes available for processing last. Now $\tau = 13 - 4 = 9$ and V={J_2, J_6}. Minimum lateness over V = min {(9-6), (9-7)}, occurs for J_6. So J_6 is scheduled fourth.

4) Find the job to be processed third:

J_3, J_5 and J_6 have been deleted from our list. J_2 becomes available for processing last. Now $\tau = 9 - 1 = 8$ and V={J_2, J_4}. Minimum lateness over V = min {(8-6), (8-7)}, which occurs for J_4. So J_4 is scheduled third.

5) The jobs scheduled first and second are now clearly J_1 and J_2, respectively, for the precedence constraints.

The final schedule is:

$J_1 \rightarrow J_2 \rightarrow J_4 \rightarrow J_6 \rightarrow J_3 \rightarrow J_5$

2.2.6 Smith's Algorithm [Smith 1956]

We know that EDD rules can solve the one-machine and n jobs, minimizing the maximum tardiness problem denoted by ($1/n/T_{max}$). But when we construct such a schedule we find $T_{max} = 0$, (i.e. all due dates can be met). Then we might think to optimize the schedule the other way. Indeed Smith's algorithm gives us a way of finding a schedule to minimize $\bar{F}$ subject to the condition that $T_{max} = 0$.

Thus for the n jobs, one machine problem when all the due dates can be met, there exists a schedule which minimizes $\bar{F}$ subject to $T_{max} = 0$ and in which job J_k is last if and only if

(a) $d_k \geq \sum_{j=1}^{n} P_j$.

(b) $P_k \geqslant P_l$ for all jobs J_l such that $d_l \geq \sum_{i=1}^{n} P_j$.

Smith's Algorithm

Step 1: Set k=n, $\tau = \sum_{j=1}^{n} P_j$; U = { $J_1, J_2, \ldots, J_n$ }

Step 2: Find $J_{j(k)}$ in U such that (a) $d_{j(k)} \geq \tau$ and (b) $P_{j(k)} \geqslant P_l$ for all J_l in U such that $d_l \geq \tau$.

Step 3: Decrease k by 1; decrease τ by $P_{j(k)}$; delete $J_{j(k)}$ from U.

Step 4: If there are more jobs to schedule, i.e. if $k \geqslant 1$, go to Step 2. Otherwise stop with the optimal processing sequence $(J_{j(1)}, J_{j(2)}, \ldots , J_{j(n)})$.

In stating the algorithm we have used the following notation:

k	the position in the processing sequence currently being filled (k cycles down n, (n-1), ..., 1)
τ	the time at which the job kth in the sequence must complete
U	the set of unscheduled jobs

Example: One machine, 4 jobs, minimizing average flow time problem denoted by $(1/4/\bar{F})$ subject to $T_{max} = 0$.

Job	J_1	J_2	J_3	J_4
Processing Time	2	3	1	2
Due Date	5	6	7	8

Step 1: k=4, $\tau = 2 + 3 + 1 + 2 = 8$, U = $\{J_1, J_2, J_3, J_4\}$.

Step 2: Only J_4 satisfies condition (a) so we choose $J_{j(4)} = J_4$

Step 3: k=3, $\tau = 6$, U = $\{J_1, J_2, J_3,\}$.

Step 4: $k \geq 1$. (return to Step 2)

Step 2: J_2 and J_3 satisfy condition (a); J_2 has the larger processing time, so $J_{j(3)} = J_2$

Step 3: k=2, $\tau = 3$, U = $\{J_{1,}, J_{3,}\}$.

Step 4: $k \geq 1$. (return to Step 2)

Step 2: J_1 and J_3 satisfy condition (a); J_1 has the larger processing time, so $J_{j(2)} = J_1$

Step 3: k=1, $\tau = 1$, U = $\{ J_{3,} \}$.

Step 4: $k \geq 1$. (return to Step 2)

Step 2: J_3 satisfies condition (a), so $J_{j(1)} = J_3$

Step 3: k=0, $\tau = 0$, U is empty.

Step 4: Optimal sequence is (J_3, J_1, J_2, J_4).

The following examination shows that EDD does not minimize $\bar{F}$ subject to T_{max}=0, but Smith's algorithm does. For the preceding example, the scheduling result of Smith's Algorithm is (J_3, J_1, J_2, J_4), and the scheduling result of EDD is (J_1, J_2, J_3, J_4):

$$\bar{F} = \frac{1}{n}\left(\sum_{j=1}^{n} W_j + \sum_{j=1}^{n} P_j \right)$$

Where:

$\sum_{j=1}^{n} W_j$ total waiting time.

$\sum_{j=1}^{n} P_j$ total processing time.

$\sum_{j=1}^{4} P_j = 8$ for both schedules

$\sum_{j=1}^{4} W_j = W_3 + W_1 + W_2 + W_4 = 0 + 1 + 3 + 6 = 10$ (Scheduling by Smith's algorithm)

$\sum_{j=1}^{4} W_j = W_1 + W_2 + W_3 + W_4 = 0 + 2 + 5 + 6 = 13$ (Scheduling by EDD)

$\bar{F} = \frac{1}{4}(10 + 8)$ (Average flow time by Smith's algorithm)

$\bar{F} = \frac{1}{4}(13 + 8)$ (Average flow time by EDD)

2.2.7 Van Wassenhov And Gelders' Algorithm [Van Wassenhov and Gelders 1980] (Finding Schedules efficient with respect to T_{max} and that $\bar{F}$)

According to Smith's theory, we are only willing to consider reducing F (mean flow time) once we have ensured that $T_{max} = 0$. In other words, penalty costs have overriding importance. Yet, if we are prepared to allow T_{max} to rise, we might be able to reduce $\bar{F}$ more than sufficiently to compensate for an increase in T_{max} . Thus, the focus now is to minimize $\bar{F}$ subject to $T_{max} \leq \Delta$, (i.e. subject to no job being finished more than

Δ after its due date). The algorithm can also be described as solving the one machine, n jobs, minimizing the average flow time problem denoted by ($1/n/\bar{F}$) subject to $T_{\max} \leq \Delta$. By adding Δ to all the due dates and apply Smith's algorithm we find:

Step 2 of Smith's algorithm can be modified and replaced by:

Step 2: Find $J_{j(k)}$ in U such that (a) $d_{j(k)} \geq \tau$ and (b) $P_{j(k)} \geqslant P_l$ for all J_l in U such that $d_l \geq \tau$. If there is a choice for $J_{j(k)}$, choose $J_{j(k)}$ to have the latest possible due date.

When this modification is made, the algorithm always finds an efficient schedule.

Based on this principle, Van Wassenhov and Gelders developed their algorithm as follows:

Step 1: Set $\Delta = \sum_{j=1}^{n} P_j$.

Step 2: Solve the $1/n/\bar{F}$ problem subject to $T_{\max} \leq \Delta$ using the modified version of Smith's algorithm. If Step 2 of that algorithm involves an arbitrary choice, repeat the solution until all possible choices have been made. If there is no schedule with $T_{\max} \leq \Delta$, go to Step 5.

Step 3: Let the schedule(s) found in Step 2 have $T_{\max} \leq \Delta_0$. Set $\Delta = \Delta_0 - 1$.

Step 4: If $\Delta \geq 0$, go to Step 2. Otherwise, continue to Step 5.

Step 5: Stop.

Example: A $1/4/\bar{F}$ problem subject to $T_{\max} \leq \Delta$

Job	J_1	J_2	J_3	J_4
Processing Time	2	4	3	1
Due Date	1	2	4	6

Step 1: $\Delta = 10$

Step 2: Find an efficient sequence (J_4, J_1, J_3, J_2) with $\bar{F} = 20$ and $T_{max} = 8$.

Step 3: $\Delta = 7$

Step 4: $\Delta \geqslant 0$.

Step 2: Find an efficient sequence (J_4, J_1, J_2, J_3) with $\bar{F} = 21$ and $T_{max} = 6$.

Step 3: $\Delta = 5$

Step 4: $\Delta \geqslant 0$.

Step 2: Find an efficient sequence (J_1, J_2, J_3, J_4) with $\bar{F} = 27$ and $T_{max} = 5$.

Step 3: $\Delta = 4$

Step 4: $\Delta \geqslant 0$.

Step 2: No sequence with $T_{max} \leqslant 4$.

Step 5: Stop.

Suppose the total cost in this example is linear with positive coefficients, say:

$$C(T_{\max}, \overline{F}) = 4T_{\max} + 7\overline{F}$$

The total cost of the three schedules are:

(J_4, J_1, J_3, J_2): $T_{max} = 8$, $\bar{F} = 20$. Total cost = 4x8 + 7x20 = 172.

(J_4, J_1, J_2, J_3): $T_{max} = 6$, $\bar{F} = 21$. Total cost = 4x6 + 7x21 = 171.

(J_1, J_2, J_3, J_4): $T_{max} = 5$, $\bar{F} = 27$. Total cost = 4x6 + 7x27 = 209.

Hence, the minimal cost schedule is (J_4, J_1, J_2, J_3)

2.3 Algorithms For Flow-Shop, Job-Shop And Open Shops Problems.

In this section, the scheduling problem is described by a four-field classification $n/m/A/B$. [Conway, Maxwell, and Miller, 1967], [Lenstra 1977], [Rinnooy Kan, 1976] and Graham et al. 1979].

We shall now discuss problems in which each job requires execution on more than one machine. From section 2.1, we know that in an open shop the order in which a job passes through the machine is immaterial, whereas in a flow shop each job has the same machine ordering (M_1, M_2,...,M_m) and in a job shop the jobs may have different machine orderings.

Very few multi-operation scheduling problems can be solved in polynomial time. The most famous cases are the $n/2/F/F_{max}$ [Johnson 1954], $n/2/O/C_{max}$ [Gonzalez and Sahni, 1976]. We will limit our survey to these algorithms and their extended applications.

2.3.1 Algorithms For Flow-Shop Problems

In many manufacturing and assembly facilities a number of operations have to be done on every job. Often, these operations have to be done on all jobs in the same order, which implies that the jobs have to follow the same route. The machines are assumed to be set up in series and the environment is referred as a flow-shop. Johnson [1954] gives an O(n log n) algorithm to solve the $n/2/F/F_{max}$ problem. The logic turns out to be simple. Johnson [1954] also provide a particular case to solve the $n/3/F/F_{max}$ problem in polynomial time, though $n/3/F/F_{max}$ is strongly NP-hard [Garey, Johnson and Sethi 1976]. Enumerative methods are also commonly used in real life because of their straightforwardness and simplicity. These three algorithms are shown in this section and simple examples are also provided in the following section.

2.3.1.1 Johnson's Algorithm For The $N/2/F/F_{max}$ Problem [Johnson 1954]

(n job, 2 machines (each job in order M1, M2), flow-shop, to minimize maximum flow time)

- The basic logic of Johnson's algorithm for the $n/2/F/F_{max}$ problem with $P_{j1} = a_j$ and $P_{j2} = b_j,\ j = 1, 2, \ldots, n$:
  1. (1) if $a_k = \min\{a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n\}$, there is an optimal schedule with J_k first in the processing sequence;
  2. (2) if $b_k = \min\{a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n\}$, there is an optimal schedule with J_k last in the processing sequence;
- Johnson's Algorithm

Step 1: Set $k=1,\ l=n$.

Step 2: Set the current list of unscheduled jobs = $\{J_1, J_2, \ldots, J_n\}$.

Step 3: Find the smallest of all the a_j and b_j times for the jobs currently unscheduled.

Step 4: If the smallest time is for J_j on first machine, i.e. a_j is smallest, then:

(1) Schedule J_j in kth position of the processing sequence.

(2) Delete J_j from the current list of unscheduled jobs.

(3) Increment k to $k+1$.

(4) Go to Step 6.

Step 5: If the smallest time is for J_j on the second machine, i.e. b_j is smallest, then:

(1) Schedule J_j in the lth position of the processing sequence.

(2) Delete J_j from the current list of unscheduled jobs.

(5) Reduce l to l -1.

(6) Go to Step 6.

Step 6: If there are any jobs still unscheduled, go to Step 3. Otherwise stop.

Note: If the smallest time occurs for more than one job in Step 3, then pick J_j arbitrarily.

Example: A $7/2/F/F_{max}$ scheduling problem. The processing time on machines is as follows:

(The problem contains 7 jobs, 2 machines, flow shop, which minimize maximum flow time).

Job	M_1	M_2
1	6	3
2	2	9
3	4	3
4	1	8
5	7	1
6	4	5
7	7	6

Applying the algorithm, the schedule is as follows:

Job 4 scheduled:	4 –						
Job 5 scheduled:	4	*	*	*	*	*	*

Job 2 scheduled:	4	*	*	*	*	*	5
Job 3 scheduled:	4	2	*	*	*	*	5
Job 1 scheduled:	4	2	*	*	*	3	5
Job 6 scheduled:	4	2	6	*	1	3	5
Job 7 scheduled:	4	2	6	7	1	3	5

Hence, the optimal order is (4, 2, 6, 7, 1, 3, 5)

In the preceding schedule, there are two arbitrary choices. We could have put Job 5 into the last position of the sequence before scheduling Job 4, and the resulting sequence would have been the same. Also we could have scheduled Job 1 in the sixth position instead of Job 3. This would have led to a different, but equivalent, processing sequence (4, 2, 6, 7, 3, 1, 5).

2.3.1.2 Johnson's Algorithm For The *N/3/F/F*$_{max}$ Problem [Johnson 1954][Szwarc1977]

Johnson's algorithm for the *n/3/F/F*$_{max}$ problem is a special case of his algorithm for *n/2/F/F*$_{max}$ problem. Here are the pre-conditions for this problem:

either

$$\min_{j=1}^{n}\{P_{j1}\} \geq \max_{j=1}^{n}\{P_{j2}\}$$

or

$$\min_{j=1}^{n}\{P_{j3}\} \geq \max_{j=1}^{n}\{P_{j2}\}$$

i.e. the maximum processing time on the second machine is no greater than the minimum time on either the first or the third. In such a case, an optimal schedule for the problem may be found by letting

$$a_j = p_{j1} + p_{j2}$$

$$b_j = p_{j2} + p_{j3}$$

and processing the jobs as if they are processed on two machines only, but with the processing time of each job being a_j and b_j on the first and second machine respectively.

Example: A $6/3/F/F_{max}$ problem with processing time and order is as follows:

Actual Processing Time				Constructed Processing Time	
Job	M_1	M_2	M_3	1st Machine	2nd Machine
1	4	1	3	5	4
2	6	2	9	8	11
3	3	1	2	4	3
4	5	3	7	8	10
5	8	2	6	10	8
6	4	1	1	5	2

(a) Check the processing time on machines 1, 2, and 3.

$$\min_{j=1}^{6}\{P_{1j}\}=3; \qquad \max_{j=1}^{6}\{P_{2j}\}=3; \qquad \min_{j=1}^{6}\{P_{j3}\}=1.$$

Thus we have $\min_{j=1}^{6}\{P_{1j}\}=\max_{j=1}^{6}\{P_{2j}\}=3$. (pre-condition is satisfied)

(b) Construct a_i and b_i times. $a_j = p_{j1} + p_{j2}$ and $b_j = p_{j2} + p_{j3}$. For the results of the equation see the preceding table.

(c) If we apply Johnson's algorithm for $n/2/F/F_{max}$ we have the following processing sequence (2, 4, 5, 1, 3, 6).

For the results of sequencing see Fig. 5 Gantt Chart for $6/3/F/F_{max}$ Problem.

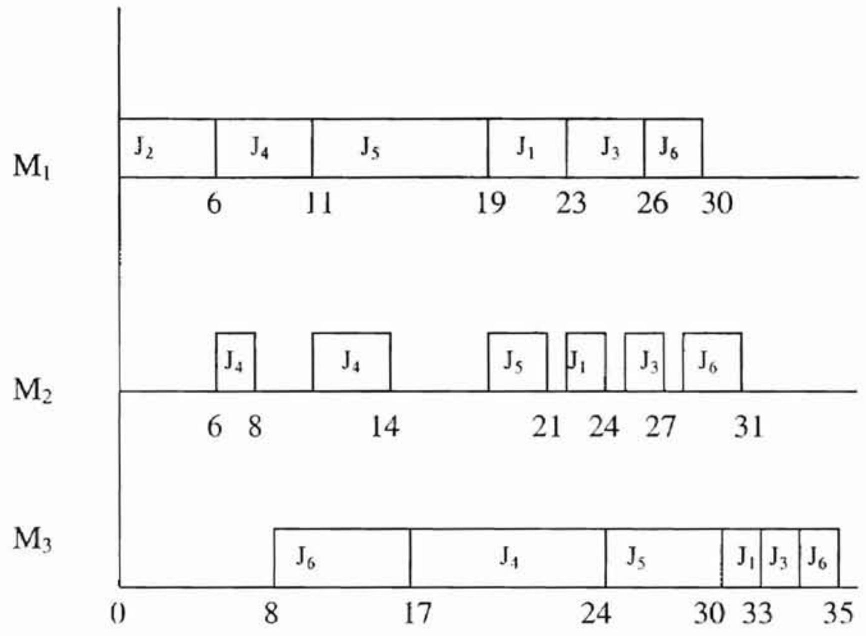

Fig. 5 Gantt Chart for the $6/3/F/F_{max}$ Problem

2.3.1.3 Akers' Graphical Solution To The $2/M/F/F_{max}$ Problem [Akers 1956]

(two jobs n machines flow-shop case, graphically minimizing maximum flow time)

The method generates schedules one by one, searching for an optimal solution. It uses procedures of elimination to see if the non-optimality of one schedule implies the non-optimality of many others not yet generated. It may not search all of the sets of feasible solutions.

Use a horizontal line to represent work on one job, a vertical line on the other, and a 45° line on both. The optimal schedule can be drawn with the following steps.

Step 1. On a piece of graph paper, lay out the processing times of job 1, in order of processing on the (X) horizontal axis. Lay out the processing times of job 2 in order of processing on the (Y) axis. (total time on job 1 $\geq$ total time on job 2).

Step 2. Find the oblong area where the processing time on the first machine required by job 1 crosses the processing time required by that same machine on job 2. Crosshatch this area. This area is the time when both jobs require the same machine.

Step 3 Complete step 2 for all remaining machines.

Step 4 Start from the origin, (0,0), draw (if possible) a 45° line until you hit an oblong area. Follow the edge of the oblong area until you can again go at a 45°[1]. Continue until you have completed all processing. (Justification: if we started at the origin and went to the right, job 1 would be done, letting job 2 wait. If we went straight up, job 2 would be done, letting job 1 wait. Therefore, a 45° line through a square indicates progress on both jobs).

Step 5 Starting from the origin (0,0), count each square through which the line passes. This is the time for the optimal schedule. Check to see which job has to wait by looking for horizontal (job 2 waits) or vertical (job 1 waits) lines.

[1] If you hit the corner of an oblong area, follow both edges, generating alternative solutions. Pick the line that gives the least processing time.

The maximum flow time of the schedule can also be counted by either:

$$F_{\max} = \sum_{j=1}^{m} P_{1j} + \text{sum of length of vertical segments of the schedule line}$$

or

$$F_{\max} = \sum_{j=1}^{m} P_{2j} + \text{sum of length of horizontal segments of schedule line}$$

where m ---- number of machines,

P_{1j} ---- processing time of job 1 at machine j

P_{2j} ---- processing time of job 2 at machine j

Example. Graph Schedule for a 2-job, 7-machine process with data as follows:

Job 1	Order of machines	A	C	E	F	B	D	G
	Processing times	1	2	2	4	3	1	1
Job 2	Order of machines	C	F	A	E	G	D	B
	Processing times	3	3	2	1	1	1	2

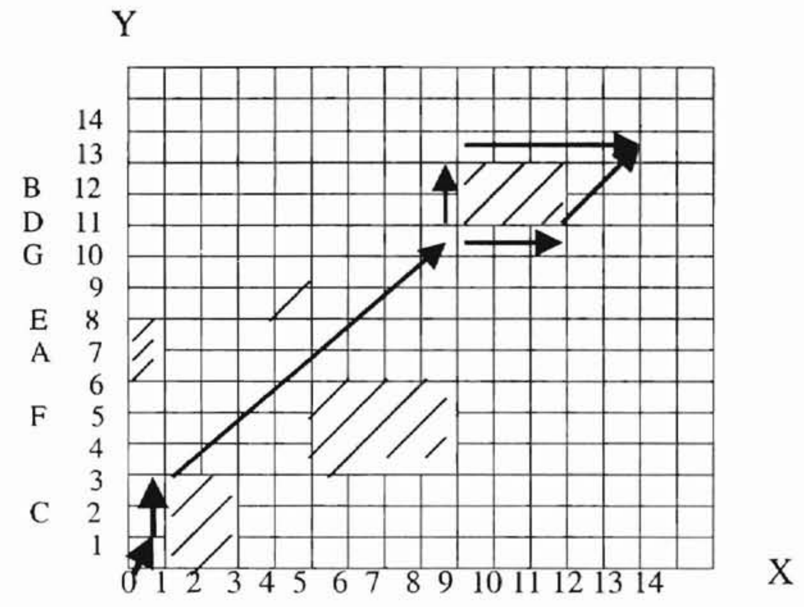

Fig. 6 Graph Of Schedule In $2/7/F/F_{max}$

2.3.2 Algorithm For Job-Shop Problems

Relaxing the flow-structure can create an immediate generalization of the flow-shop problems. Rather than requiring each job to progress through the processing stage in an identical fashion, we now allow jobs to have different ordering requirements. In this context, we also allow job operations to involve repetitious processing. By a modification of Johnson's algorithm for $n/2/F/F_{max}$, the $n/2/G/F_{max}$ (n job, 2 machine general job-shop, minimizing maximum flow time) the problem can be solved in polynomial time.

Johnson's algorithm for the $n/2/G/F_{max}$ problem [Johnson 1954] is as follows:

Suppose that the set of n jobs $\{ J_1, J_2, ..., J_n\}$ may be partitioned into four types of jobs as follows:

Type A: those to be processed on machine M_1 only.

Type B: those to be processed on machine M_2 only.

Type C: those to be processed on both machines in the order M_1 then M_2.

Type D: those to be processed on both machines in the order M_2 then M_1

The construction of an optimal schedule is straightforward.

(1) Schedule the jobs of type A in any order to give the sequence S_A.

(2) Schedule the jobs of type B in any order to give the sequence S_B.

(3) Schedule the job of type C according to Johnson's algorithm for $n/2/F/F_{max}$ problems to give the sequence S_C.

(4) Schedule the job of type D according to Johnson's algorithm for $n/2/F/F_{max}$ problems to give the sequence S_D (here M_2 is the first machine and M_1 is the second machine).

An optimal schedule is then:

Machine	Processing Order
M_1	(S_C, S_A, S_D)
M_2	(S_D, S_B, S_C)

This schedule clearly minimizes the idle time when M_2 is kept waiting for jobs of Type C to complete on M_1 or when M_1 is kept waiting for jobs of Type D to complete on M_2. Therefore it is an optimal schedule.

Example: A $9/2/G/F_{max}$ problem with times and processing order as follows:

Job	First Machine		Second Machine	
1	M_1	8	M_2	2
2	M_1	7	M_2	5
3	M_1	9	M_2	8

4	M_1	4	M_2	7
5	M_2	6	M_1	4
6	M_2	5	M_1	3
7	M_1	9	*	
8	M_2	1	*	
9	M_2	5	*	

Find the optimal schedule.

Type A jobs: Only job 7 is to be processed on M_1 alone.

Type B jobs: Jobs 8 and 9 require M_2 alone. Select arbitrary order (8, 9).

Type C jobs: Jobs 1, 2, 3, and 4 require M_1 first and then M_2. Johnson's algorithm for this $4/2/F/F_{max}$ problem gives the sequence (4, 3, 2, 1).

Type D jobs: Jobs 5, 6 require M_2 first and then M_1. Johnson's algorithm for this $2/2/F/F_{max}$ problem gives the sequence (5, 6)(M_1 is the second machine).

Hence, an optimal sequence for the overall problem is:

	Job Processing Sequence
Machine M_1	(4, 3, 2, 1, 7, 5, 6)
Machine M_1	(5, 6, 8, 9, 4, 3, 2, 1)

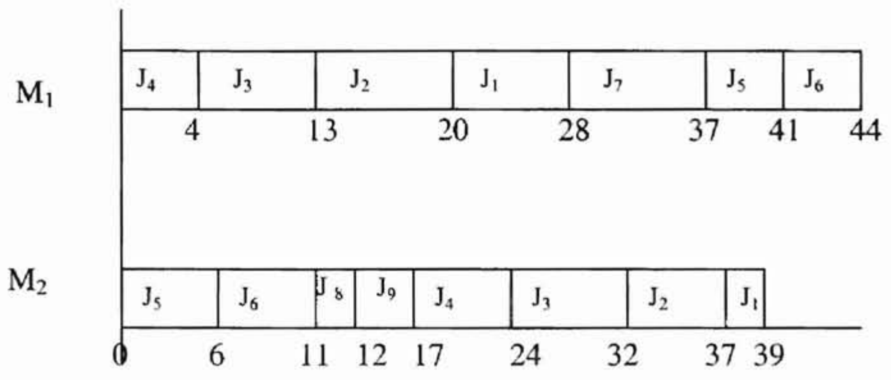

Fig. 7 Gantt Chart For The $9/2/G/F_{max}$ Problem

2.3.3 Algorithm For Open Shop Problems

Though open shop problems have been proved to be NP-Complete [Parker 1995], minimizing makespan in the open shop with two machines ($n/2/O/C_{max}$) problem turns out to be solvable in polynomial time due to Gonzalez and Sahni's [1976] contribution.

Algorithm for n jobs, two machines open shop problem ($n/2/O/C_{max}$).

Step 1: Initialize by setting $\Delta_1 = \Delta_2 = k = r = P_{01} = P_{02} = 0$; $\pi = 0$; $i = 0$;

Step 2: Compute $\Delta_1 \leftarrow \Delta_1 + P_{i1}$; $\Delta_2 \leftarrow \Delta_2 + P_{i2}$;

If $P_{i1} \geq P_{i2}$, go to step 3, else go to Step 4;

Step 3: If $P_{i1} \geq P_{i2}$, then extend π by concatenating as πr and set $r \leftarrow i$; else, concatenate as πi. If $i = n$, go to Step 5; else set $i \leftarrow i+1$ and return 2.

Step 4: If $P_{i2} \geq P_{k1}$, then extend π by concatenating as $k\pi$ and set $k \leftarrow i$; else, concatenate as $i\pi$. If $i = n$, go to Step 5; else set $i \leftarrow i+1$ and return 2.

Step 5: If $\Delta_1 - P_{k1} < \Delta_2 - P_{k2}$, set $\pi_1 \leftarrow \pi rk$ and $\pi_2 \leftarrow k\pi r$; else set $\pi_1 \leftarrow k\pi r$ and

$\pi_2 \leftarrow rk\pi$. Remove all 0's from the permutations π_1 and π_2 and schedule in these orders on respective processors.

Example: Four jobs, two machines, open-shop, minimizing makespan problem ($4/2/O/C_{max}$). The four jobs and operation times are shown bellow:

Job	M_1	M_2
1	7	3
2	2	4
3	5	8
4	2	6

The computation can be easily summarized in the following table:

After iteration	π	r	k
1	0	1	0
2	00	1	2
3	200	1	3
4	3200	1	4

By applying Step 5, we form $\pi_1 \leftarrow \pi rk = \{320014\}$ and $\pi_2 \leftarrow k\pi r = \{432001\}$. Or upon eliminating dummy 0s, we obtain $\pi_1 = \{3214\}$ and $\pi_2 = \{4321\}$. The corresponding schedule is shown in Fig. 8.

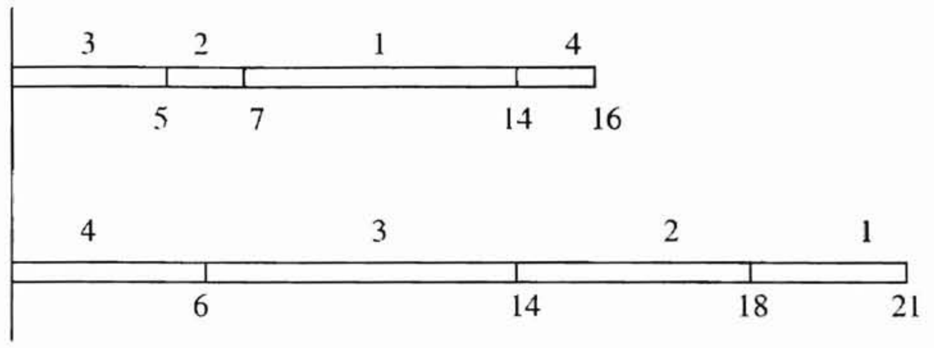

Fig. 8 Final Schedule for $4/2/O/C_{max}$.

2.4 Algorithms For Parallel Machine Problems

In this section, problems are described by a three-field classification $\alpha/\beta/\gamma$. [Conway, Maxwell, and Miller, 1967], [Lawler, Lenstra, and Rinnooy Kan, 1982], [Herrmann, Lee and Snowdon, 1993].

For parallel machine problems, we assume an environment characterized by two or more machines where these machines have similar capabilities (but perhaps different performance qualities, such as speed and so on). That is, any of $M \geqslant 2$ machines are available for processing a given job. The aim is to find an assignment of all jobs to existing machines that makes optimal some predetermined measure. Here we focus on algorithms for the makespan case.

This problem can be described as a given finite set of jobs J, a non-negative duration t_i for each $i \in J$, a number $m \geq 2$ of machines, and a completion time threshold $D>0$, find a partition $(J_1, J_2, ,\ldots, J_m)$ of J such that

$$\max\{\sum_{i\in J_k} t_i; 1 \le k \le m\} \le D$$

It is clear that for any fixed number of m machines, the problem can be solved in polynomial time. But when m is free, it is a NP-complete problem. Facing this difficulty, it comes as no surprise that the problem has been studied in the context of approximation approaches.

2.4.1 List Processing [Parker, R. G. 1995]

The method can be summarized in the following manner:

Create a list of jobs L and from this list, form a schedule as follow. Whenever a processor becomes available, schedule the first available job from the list.

The building of the list *L* might be guided by some sense of priority among jobs with the latter based on attributes such as job duration, due dates, and so on. The schedule building is simple and the results of the schedule satisfy the user's required priority, but the schedule is not necessarily optimal.

Example: A three-machine problem with duration time and ordered list as follows:

Job Processing Time and Order List

Processing time:	7	3	2	5	4	8	6	3	5	4
Ordered List:	1	2	3	4	5	6	7	8	9	10

Following the preceding rules, the lists generated are (1,6,), (2,5,7,10), (3,4,8,9). See Fig. 9.

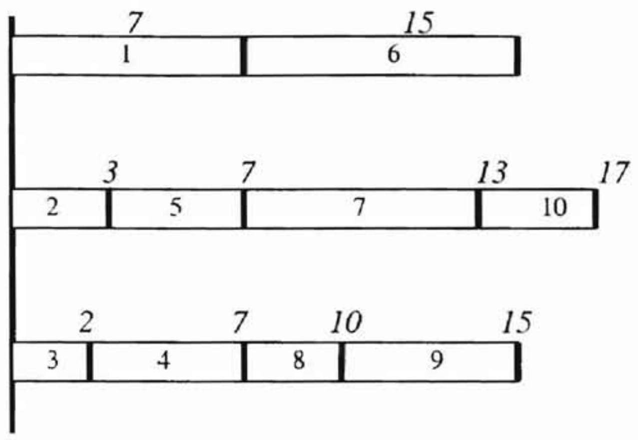

Fig. 9 List Processing Schedule

2.4.2 Longest Processing Time Heuristic [Graham 1969]

The method can be summarized as follows:

Create L with jobs arranged in non-increasing t-order (longest processing time or LPT order). Form a schedule as follows: whenever a processor becomes available, schedule the first available job from the list.

Example: A four-machine problem with duration time and ordered list as follows:

Job Processing Time and Order List

Processing time:	7	7	6	6	5	5	4	4	4
Job Order List:	1	2	3	4	5	6	7	8	9

Following the preceding rule, the list processing generated is (1, 7, 9), (2, 8), (3, 5), (4, 6). The result is shown in Fig. 10.

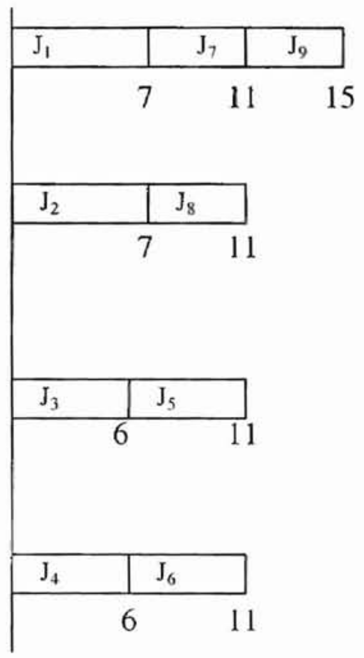

Fig.10 Example for Longest Processing Time Heuristic

2.4.3 MULTIFIT Heuristic [Coffman et al. 1978]

For some time the Longest Processing Time Heuristic was the best known in terms of a performance guarantee. However, this position of prominence was relinquished when Coffman et al. (1978) offered the MULTIFIT Heuristic which cleverly employs a natural dual-like relationship between the problem $P||C_{max}$ and the well-known BIN-PACKING problem.

BIN-PACKING seeks an admissible assignment or packing of a finite set of "chips," each with some positive weight, into the fewest number of finite capacity "bins." This dualistic relationship should be evident as illustrated by the following:

BIN-PACKING	$P\|\|C_{max}$
Bins	Machines
Capacity	C_{max} threshold
Chips	Jobs

Moreover, if packing into no more than m bins each with capacity of C, then there exists a suitable schedule with makespan no greater than C.

BIN-PACKING logic is considered for the following heuristics:

2.4.3.1 First-Fit, Decreasing Weight Heuristic (FFD)

Create a list L of chips arranged in non-increasing weight-order. Select chips from L in this order, placing a given selection in the first available bin into which it will fit.

Example: Let C = 61 and consider the list of chips given as L = (44, 24, 24, 22, 21, 17, 8, 8, 6, 6). Applying the First-Fit, Decreasing Weight Heuristic produces the four-bin packing in Fig.11.

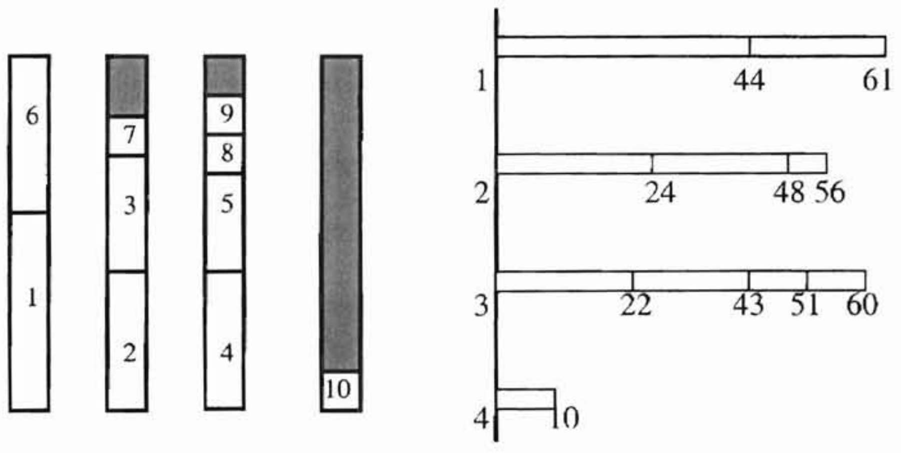

Fig. 11 First-Fit, Decreasing Weight Heuristic

2.4.3.2 MULTIFIT Heuristic

Step 1: Initialization. Let T be the set of jobs and fix upper and lower bounds relative to T and m, as $\beta_U[T,m]$ and $\beta_L[T,m]$ respectively. Let $\beta_1(0) \leftarrow \beta_U$ and $\beta_2(0) \leftarrow \beta_L$. Choose a number of iterations t and an iteration counter $i \leftarrow 1$.

Step 2: Capacity change. If i > t, stop. Otherwise, set

$$C \leftarrow (\beta_2(i-1) + \beta_1(i-1)) \Big/ 2$$

Step 3: Upper bound. If the number of bins required by the First-Fit, Decreasing Weight Heuristic operating on T with capacity C, given as FFD[T,C], is no greater than m, set $\beta_1(i) \leftarrow C$, $\beta_2(i) \leftarrow \beta_2(i-1)$, update $i \leftarrow i+1$ and go to Step 2.

Step 4: Lower bound. If FFD[T,C]> m, set $\beta_2(i) \leftarrow C$, $\beta_1(i) \leftarrow \beta_1\ (i-1)$, update $i \leftarrow i+1$ and go to Step 2

where

$$\beta_L[T,m] = \max\left\{\sum t_j \Big/ m, \max_j(t_j)\right\}$$

$$\beta_U[T,m] = \max\left\{2\sum t_j \Big/ m, \max_j(t_j)\right\}$$

Example. A three-machine, seven-job problem with data as follows:

Job Number:	1	2	3	4	5	6	7

Processing time: 59 47 38 22 13 12 11

Choose iteration t = 6.

The fixed Lower bound is:

$$\beta_L = (59 + 47 + 38 + 22 + 13 + 12 + 11) / 7 = 67.3$$

The fixed upper bound is:

$$\beta_u = 2 \times 67.3 = 134.6$$

The computation is summarized as:

When i = 1:

$C = 100.9$, $FFD[T, C] \leq 3$, $\beta_l(1) = 100.9$, $\beta_2(1) = 67.3$

When i = 2:

$C = 84.1$, $FFD[T, C] \leq 3$, $\beta_l(2) = 84.1$, $\beta_2(2) = 67.3$

When i = 3:

$C = 75.7$, $FFD[T, C] \leq 3$, $\beta_l(3) = 75.7$, $\beta_2(3) = 67.3$

When i = 4:

$C = 71.5$, $FFD[T, C] \leq 3$, $\beta_l(4) = 71.5$, $\beta_2(4) = 67.3$

When i = 5:

$C = 69.4$, $FFD[T, C] > 3$, $\beta_l(5) = 71.5$, $\beta_2(5) = 69.4$

When i = 6:

$C = 70.5$, $FFD[T, C] \leq 3$, $\beta_l(6) = 70.5$, $\beta_2(6) = 69.4$

When i = 7: Stop.

Final packing yields the schedule shown in Fig. 12.

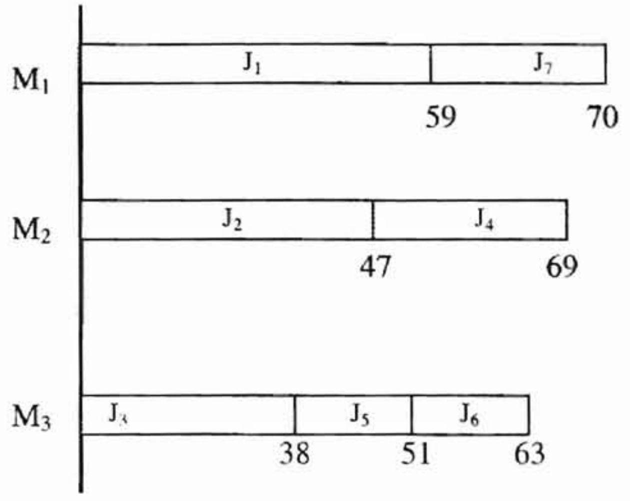

Fig. 12 Schedule of MULTIFIT Heuristic

Scheduling of jobs on parallel machines is very hard to solve. The scheduling of jobs with precedence constraints on parallel machines is harder, but there are some interesting solvable unit duration time cases. One is given by Hu [1961].

2.4.4 Hu's Algorithm

Step 1: Compute the length of a longest path from each vertex. Call these values *li*

Step 2: Create a list L of jobs arranged in non-increasing *l-order.* Perform List Processing on L.

Hu's Algorithm solves $P\backslash tree,\ t_i = 1|C_{max}$

Example: Three machines, 17 jobs are constrained as shown in Fig. 13. Each job is of one unit processing time. The final schedule can be seen in Fig. 14.

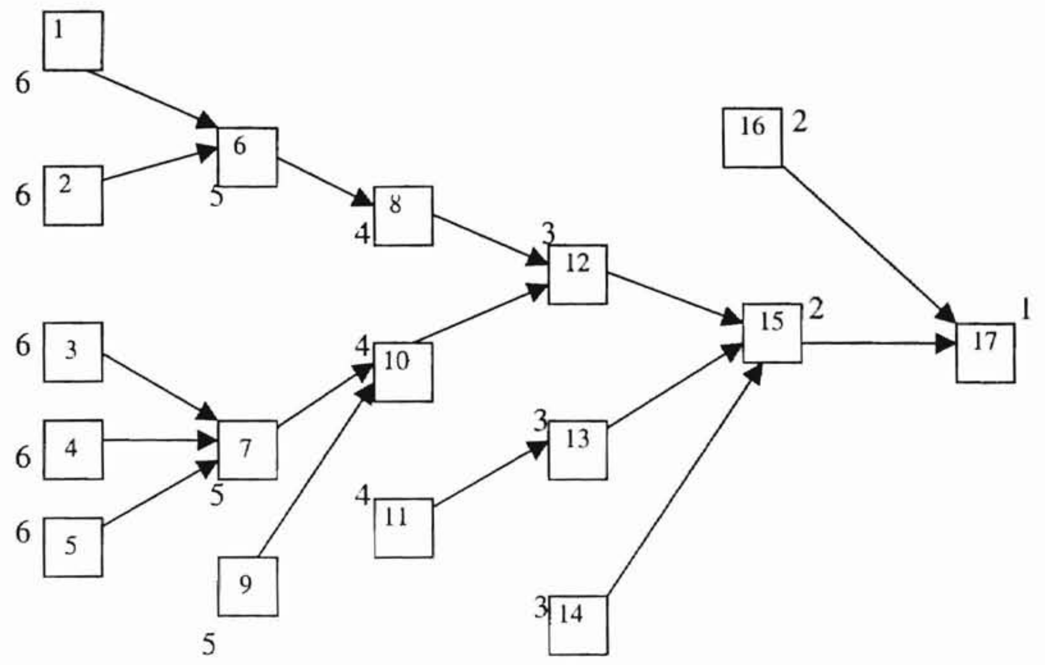

Fig. 13 Precedence Constraint Diagram

List L is built as (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17)

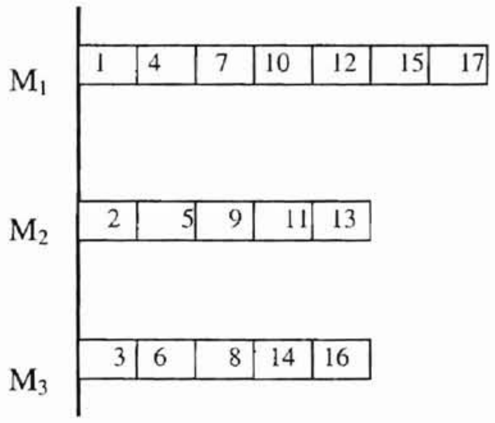

Fig. 14 Schedule of Hu's Algorithm

2.5 Some Problems In Combinatorial Optimization

Some scheduling problems can be solved efficiently by algorithms presented in the previous sections. Others require methods in combinatorial optimization. In this section, we will review the widely used methods for scheduling, such as traveling salesman and genetic algorithms.

2.5.1 Christofides Heuristic Algorithm For Traveling Salesman Problems

The traveling salesman problem (TSP) is one of the most popular problems in combinatorial optimization. Since the problem is so widely studied with most major results readily accessible in the literature, we will briefly review the Christofides' heuristic for the TSP and explicitly show this algorithm with an example.

The classic TSP asks for a tour through n "cities" that cover least total travel distance. The tour must begin and end at the same city with no city visited more than once. In graph-theoretic language, the problem is usually defined on a complete graph where edges are weighted as w: $E \rightarrow Z$ and the aim is to find a spanning cycle of the least amount of total weight. Christofides' heuristic is described by the following steps:

Step 1: Find a minimum weight spanning tree in G. Let this tree be given by $T \subseteq E$.

Step 2: Let the odd-degree vertices in the tree of Step 1 be denoted by V_0 and find a minimum weight perfectly matching in the subgraph induced by V_0. Let this matching be denoted by $M \subseteq E$.

Step 3: The graph formed as $M \cup T$ is called Eulerian[1]. Produce the corresponding (Eulerian) cycle and, interpret it as a vertex sequence, form a TSP tour by beginning as the initial vertex, and proceed in order, "shortcut" past duplicated vertices until the starting vertex is reached again.

Note: [1] ---- Given that the Eulerian graph is $G = (V, E)$, does G posess a walk that begins and ends at the same vertex and includes each edge exactly once? Such a walk is called an Eulerian traversal and a graph that admits it is called Eulerian [Euler 1736].

For example, consider the graph in Figure 15. The problem is shown in part G. In part a, the tree is given. The matching of Step 2 is shown in part b, and in parts c and d, the Eulerian cycle and generated tour are shown respectively.

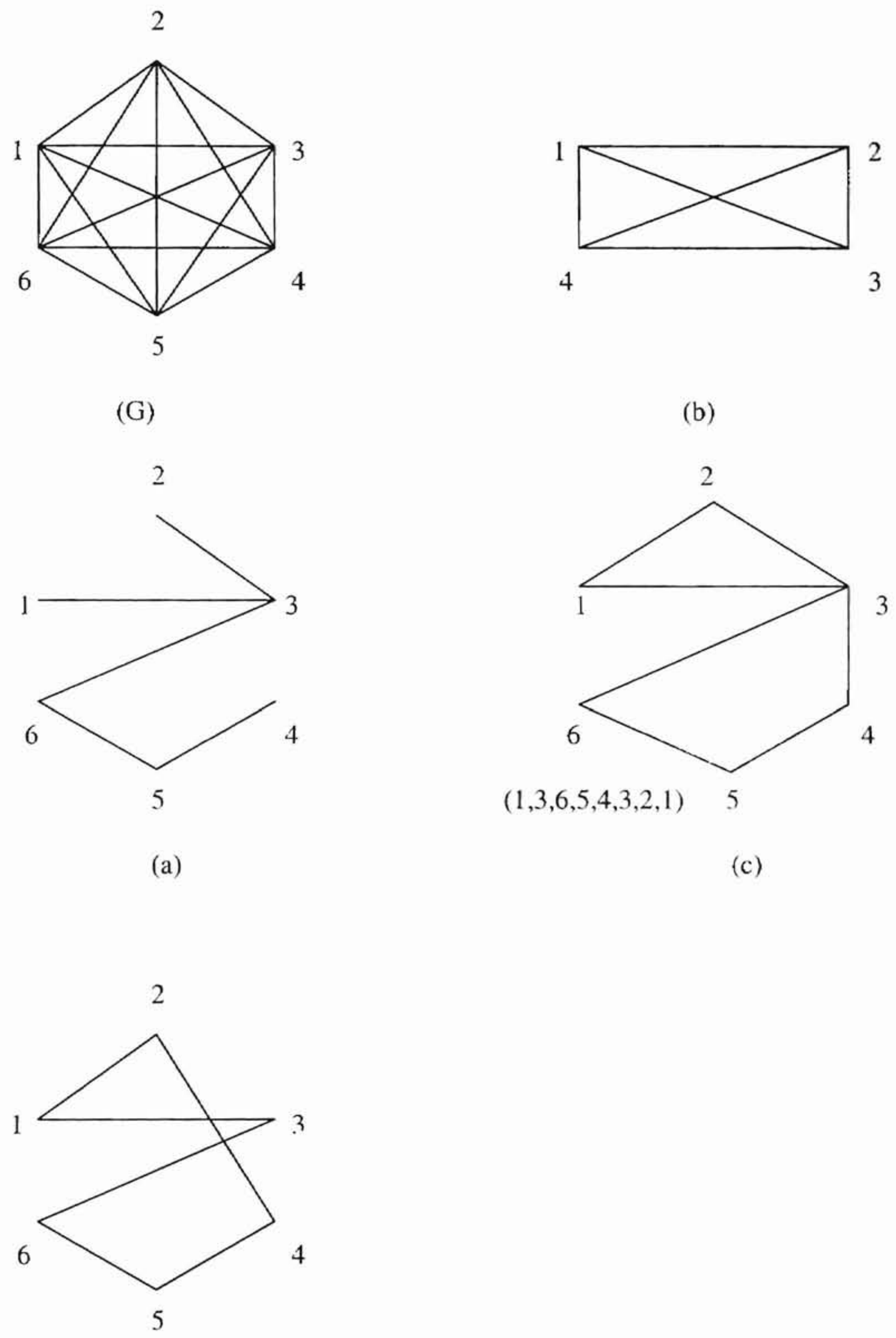

Fig. 15 Example Of Heuristic For TSP

2.5.2 Genetic Algorithms

Genetic algorithms are search algorithms based on the mechanics of natural selection and natural genetics. They combine survival of the fittest among string structures with a structured yet randomized information exchange to form a search algorithm [Goldberg 1989]. Genetic algorithms have been widely used in search and optimization in the areas of biology, computer science, engineering and operations research, job scheduling, and so on [Goldberg 1989], [Brown and Scherer 1995], [Kerr and Szelke 1995].

The mechanics of a genetic algorithm are surprisingly simple. It involves nothing more complex than copying strings and swapping partial strings. Its operation is composed of three operators, Reproduction, Crossover and Mutation. Reproduction is a process in which individual strings are copied according to their objective function values. After the newly reproduced strings in the mating pool are mated at random, an integer position k along the string is selected uniformly at random between 1 and the string length less than one (1, L-1). Swapping all characters between position k+1 and L inclusive creates two new strings. Mutation is a random walk through the string space for an occasional alteration of the value of a string position. Genetic algorithms are powerful in solving all kinds of problems in real life with the suitable implementation of these three operations.

Example: Maximize the function $f(x)=x^2$, where x is permitted to vary from 0 to 31.

Step 1: Choose an initial population by four repetitions of five coin tosses where head = 1, tail = 0. See table 1.

Step 2: Reproduce a new population of four strings by using random selection.

Step 3: Use coin tosses to pair off the happy couples. First couple 01101 and 11000, second couple 01000 and 10011.

Step 4: Use coin tosses to select the crossing site to cross over the mated string couples. First couple at position 4, second couple at position 2.

Step 5: Randomly select a mutation position, perform bit-by-bit mutation according to probability of mutation, (in this case no bits undergo mutation during a given generation for the rate is too small).

Step 6: If not satisfied with the result, go to Step 2, else go to Step 7.

Step 7: Stop.

The operation stops after 2 iterations. The maximum value of $f(x)=x^2$ increased from 576 to 729. See Table 1 and Table 2 for the results.

Table 1 Initial Genetic Productions.

String No.	Initial Population	x value	$f(x)$ x^2	Fitness slot size	Count
1	0 1 1 0 1	13	169	0.14	1
2	1 1 0 0 0	24	576	0.49	2
3	0 1 0 0 0	8	64	0.06	0
4	1 0 0 1 1	19	361	0.31	1

Table 2 Result of Genetic Cross Over.

Mating Pool After Prod.	Mate	Crossover Site	New Population	x value	$f(x)$ x^2
0 1 1 0 1	2	4	0 1 1 0 0	12	144
1 1 0 0 0	1	4	1 1 0 0 1	25	625
0 1 0 0 0	4	2	1 1 0 1 1	11	729
1 0 0 1 1	3	2	1 0 0 0 0	16	256

CHAPTER 3

INDUSTRIAL APPLICATIONS

Although scheduling algorithms originated from industry and have been developed to solve all kinds of problems, these algorithms can't cover all real world problems. In industrial practice we must analyze our problems, find the proper algorithm or combine a few algorithms together, or simply borrow some ideas from the available algorithms to find a good solution to our problems. Let's examine a mobile workforce dispatching problem and find a combinatorial solution to the problem.

3.1 Background

A company called CGV has hundreds of service stations scattered all over a state on the eastern coast of the United States. These services have been providing repair and maintenance services for customers in their service areas. CGV plans to merge all the small services into a few service centers. Since the maintenance technicians are scattered around the state, it is impossible to have all the technicians gather at one service center, pick up their maintenance orders, and drive back to work each morning. Therefore, CGV plans to build a computer-aided dispatching station (CDS) and furnish a laptop computer for each technician's maintenance vehicle. The CDS station will communicate with each technician through a wireless network (see Fig. 16). Thus, the technician will receive his maintenance orders for a day as soon as he gets in the van and starts his computer. The technician is required to send back his maintenance report and time sheet when a job is done or when he completes his assignment for a day. Not only the company benefits from doing this, but also customers and the technicians.

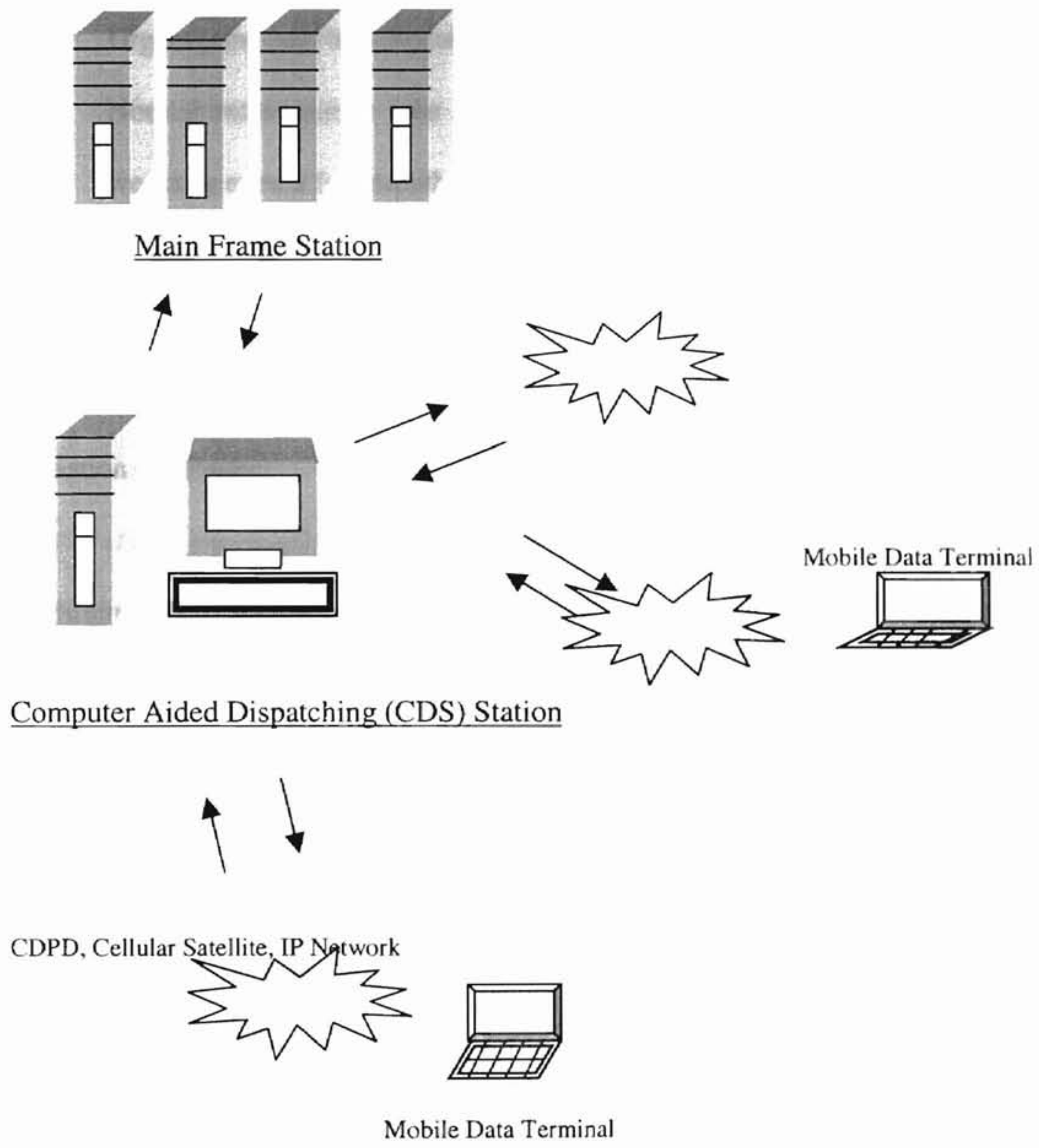
Main Frame Station
Computer Aided Dispatching (CDS) Station
Mobile Data Terminal
CDPD, Cellular Satellite, IP Network
Mobile Data Terminal

Fig.16.Network Configuration

Based on this specific condition, the company provided a short list of requirements for the design of the CDS system:

1) Every individual is assigned a responsible area.
2) Don't let technicians work overtime, if possible.
3) Customers don't like a technician to leave a job half done once it gets started.
4) Urgent jobs shall be scheduled and processed on the required date.
5) Normal maintenance jobs may be planned in the service center a few weeks or even a few months ahead of actual scheduled time.
6) Generally, the processing time ranges about 15 minutes to four hours (mostly less than one hour).

3.2 Initiation Of Project

3.2.1 Preliminary Assumptions

According to the above-mentioned requirements, the following assumptions are made to simplify the selection of scheduling algorithms:

1) All technicians are equally good at all assigned jobs.
2) If a technician doesn't have enough jobs to do in his respective region, he can request to work in nearby areas.
3) Before dispatching jobs to a technician, the service center is supposed to have received a work request that includes the technician's name, work

location, starting time and working hours of the day (a technician can make more than one request for a day).

4) The dispatching algorithm will only dispatch jobs available at dispatching time.

5) A technician can't be interrupted once he is working on a job (i.e., this is non-interruptible scheduling).

6) The sum of a technician's work time and travel time shall be within eight hours per day.

7) Special cases not mentioned above will be handled through special dispatching channels (not to be discussed in this thesis).

3.2.2 Design Of Scheduling Logic

According to the user's requirements and our assumptions, this problem seems to be a problem of m machines in parallel, n jobs with precedence constraints, minimizing total cost. Now we divide the state into a few regions. According to their locations, characterize all jobs by zip codes (i.e. group all jobs into the divided regions). Each technician is assigned to work in one region. Since these technicians' daily tasks are not closely related, we can consider one technician and n jobs as a scheduling unit. The problem thus becomes a single machine, n job problem. The parallel operation of the m technicians is separated from the scheduling of n jobs to a single technician. This parallel operation is easily managed by the CDS by checking whether or not each region has a technician logged in and jobs are evenly distributed among these regions. We only have the problem of scheduling n jobs with precedent constraints to a technician.

3.2.2.1 Job Storage And Management

1) Jobs are arbitrarily stored in the CDS database characterized by job identification number, job type (urgent or normal), estimated working time, due date, relative coordinates, address, and so on.
2) Jobs shall be dispatched from central lists to each technician's urgent job queue and normal job queue according to their due dates.
3) Priority rules for job dispatching:

- Urgent jobs have priority to be dispatched first, and normal jobs will not be dispatched unless the urgent jobs have been properly dispatched.
- On the due date of a normal job, it is automatically changed to an urgent job and transferred to the urgent list.
- If urgent jobs are in regions where there are no technicians on duty, these jobs shall be dispatched through a special channel.

3.2.2.2 Location Checking Method

The relative coordinates of each job specify all job locations. The coordinates of his registered home address also specify a technician's start point.

3.2.2.3 Procedures of Assigning Jobs To Technicians

When a technician logs in and requests some hours of work (the ready time of each technician may be different), the CDS shall send jobs to a technician according to his expected work area, available hours, and taking the technician's registered address as the start point. Following the optimality and evaluation rules, the system will dispatch jobs from his job pool, report to CDS when jobs are done. The CDS will automatic update its database with information received from each technician.

3.2.2.4 Optimality And Evaluation

The optimality of this non-interruptible scheduling is to have the technician's work time in a day as near to eight hours (may change according to his requested work hours) as possible with as little travel time as possible. Since the scheduling logic is leading to a one-machine n jobs with precedence constraints, maximizing working hours problem, this problem can be solved by applying the following algorithms repeatedly in two cases:

- Shortest travel distance first scheduling (SDF).
- Shortest process time first scheduling (SPT).
- Longest processing time first scheduling (LPT).
- Genetic SDF scheduling (GSDF).
- Genetic SPT scheduling (GSPT).
- Genetic LPT scheduling (GLPT).

- Dispatch one job from one of the above six queues that schedules the longest working hour and least traveling time in the following sense.

Case 1: Requested work time is longer than the total work time of jobs available.

The technician's work time bin is larger than all available jobs could fill. All the algorithms listed above will pack the time bin with the same amount of work time and different drive time. Since the SDF algorithm will build schedules with the shortest travel time, the system will only invoke SDF scheduling. The best solution is guaranteed.

Case 2: Plenty of jobs waiting to be processed.

Since there are plenty of jobs waiting to be processed, the technician's time bins can be packed with the six algorithms for each job request. The system will choose one queue that has been filled with the longest work time and dispatch the first job from the queue. When the job is done, the system will mark the job status "done" and transfer it to CDS' finished job database. If the technician requests another job, the system will fill the new time bin provided by the technician with remaining jobs in the job pool. The above steps will be repeated for each job request until the technician wants to stop. Since the time bin size is relatively fixed for each job request, more work time means less drive time. Each time a technician requests a job, the system will build six queues for selection. Choosing one job every time from the queue identified by maximum work time optimality is surely enough to guarantee a very good schedule.

3.2.2.5 Program Flowchart

According to the design of the scheduling logic, the following program flowchart provides us with the solution. See Fig. 17.

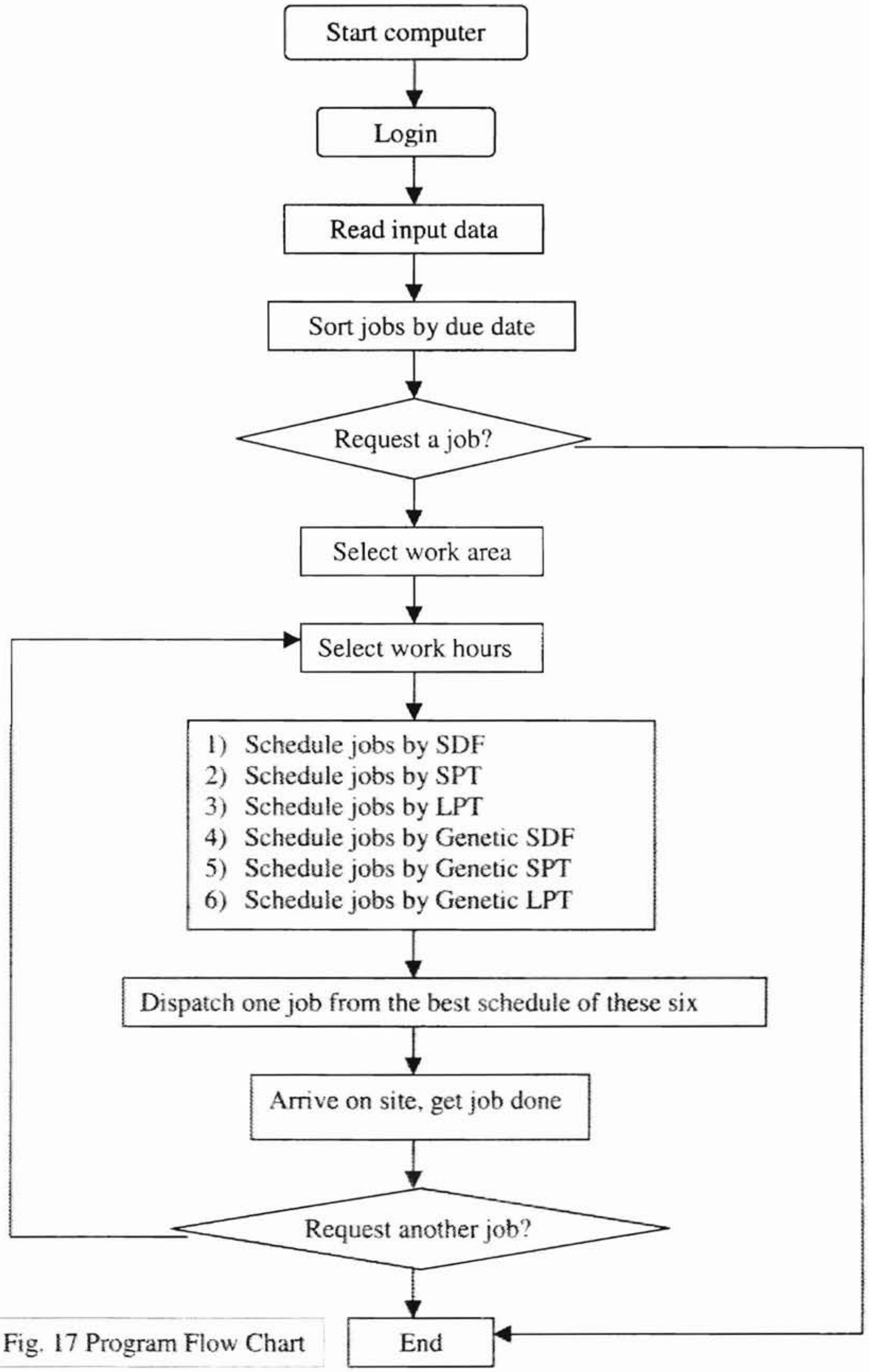

Fig. 17 Program Flow Chart

3.3 Empirical Application of the Combinatorial Solution

3.3.1 The Simulation Program

To simulate the scheduling, the following requirements have been kept in mind during programming:

a) Login and logout date and time zone are Eastern Time as provided by the computer.

b) Actual work time, travel time, mileage, jobs completed shall be reported to the CDS as soon as a job is completed.

c) The final report shall include today's total work time, mileage and jobs completed.

d) A technician can change his expected work hours when he requests another job.

e) A technician can login and logout as many times as he needs to in a day.

The program was written in C. See Appendix A.

As illustrated above, the program first employs three algorithms (SDF, SPT, LPT) to make three schedules. It is known that the three algorithms will not provide an optimal solution each time a job is dispatched. There are many uncertain conditions not considered by the three. Thus, a genetic algorithm is applied to optimize these three schedules respectively, and the result is exciting. The genetic algorithm reschedules the three queues about 6! times each (It is estimated that a technician will work out 6 jobs each day on average. The possible permutation of jobs in the queue is estimated to be 6!).

The best schedule is saved for final comparison. The final schedule chooses one job from one of these six schedules whenever a technician requests a job. Therefore, a combinatorial schedule is automatically built up, and provides us with a very good job dispatching sequence.

3.3.2 Empirical Application

In real life, there are many aspects that influence scheduling. The combinatorial scheduling logic has given certain considerations to the boundary conditions. The empirical application considers four technicians working in four different areas respectively (see Fig.18). All the job locations originated from random numbers and the technicians' home locations are arbitrarily chosen.

3.3.2.1 An Eight Hour Day at Normal Jobs

Technician Mike Jackson ID #111 requests to work eight hours in his home area with zip code 100. The system finds that normal jobs #10 to #26 in this area are available. According to his requested work area, zip code, and work hours, the system first sorts these jobs by due date and then schedules six queues of jobs each time he requests a job. The system automatically finds one job from the best of the six schedules and assigns it to the technician. When he arrives on site, the actual work time starts. When the job is done, the job is reported to the CDS. The system will ask if he requests another job, if he wishes to change expected work time, and so on. until the total time is near to eight hours. Let's examine the scheduling and assignment in detail.

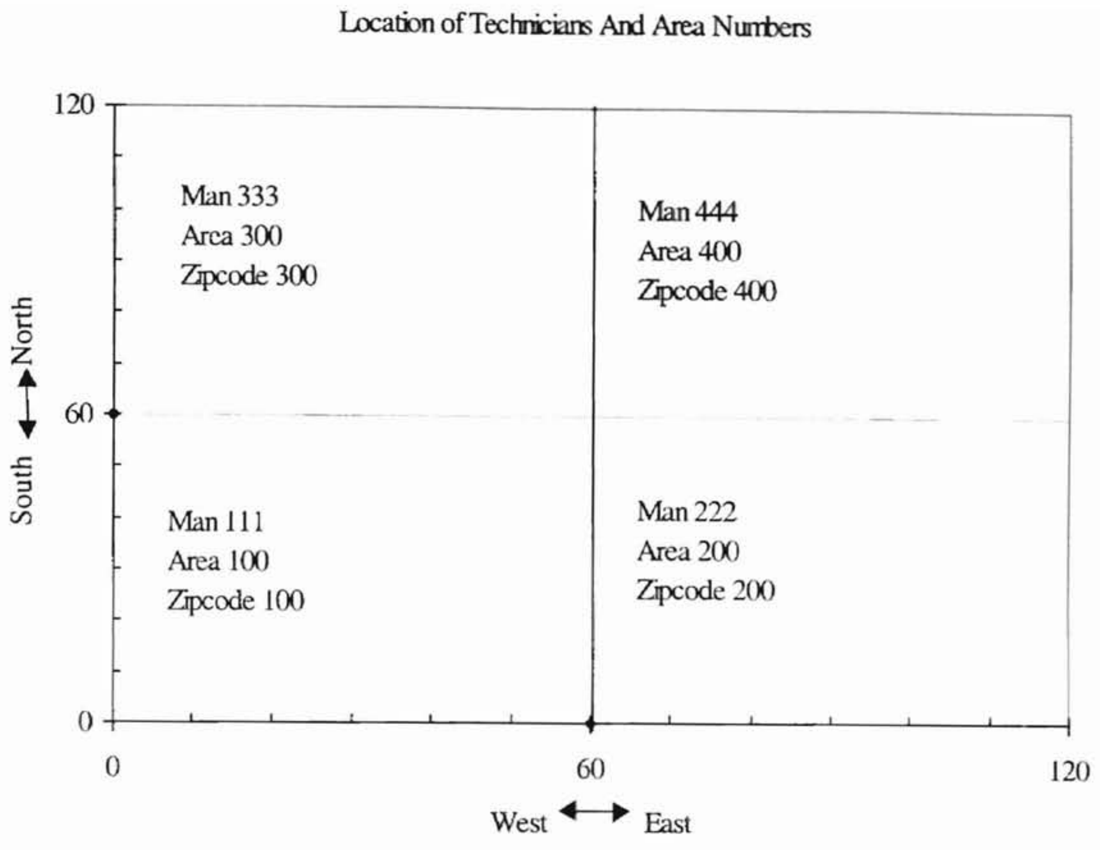

Fig. 18 Location of Technicians And Area Numbers

1) Man 111 first requests to work for eight hours. The following six queues are created.

Table 3. Eight-Hour Schedules for M111 In Area 100

Algorithm	Schedule	Work Time (hr)	Travel Time (hr)	No. of Jobs
SDF	#10, #11, #12, #15, #13, #14, #16, #17	4.8	2.4	8
SPT	#16, #19, #20, #24, #10, #11, #12, #15	3	4.26	8
LPT	#26, #25, #13	3.75	3.83	3
GSDF	#10, #11, #12, #15, #13, #14, #16, #17	4.8	2.4	8
GSPT	#11, #15, #12, #13, #14, #18, #17	4.8	3.13	7
GLPT	#10, #11, #14, #17, #18, #26	4.8	2.3	6

See Table 3 for detail schedules. These travel routines , see Fig.19 to Fig. 24..

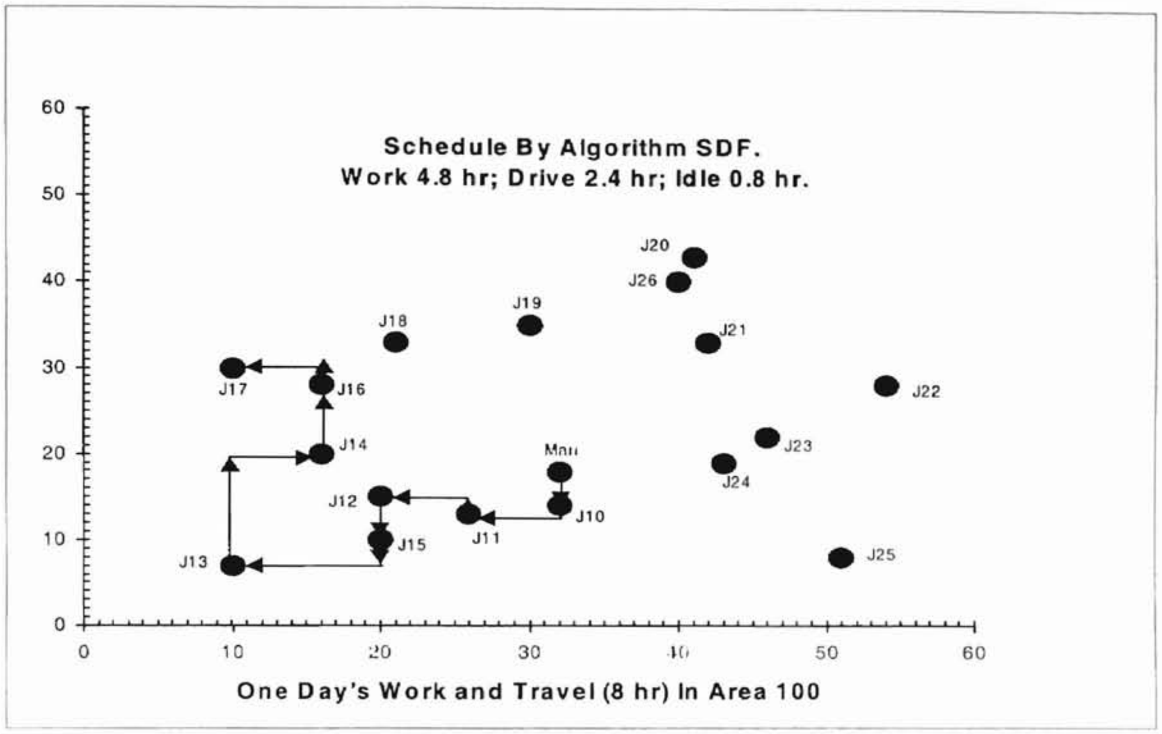

Fig. 19 Possible Travel Route for Man 111 By SDF

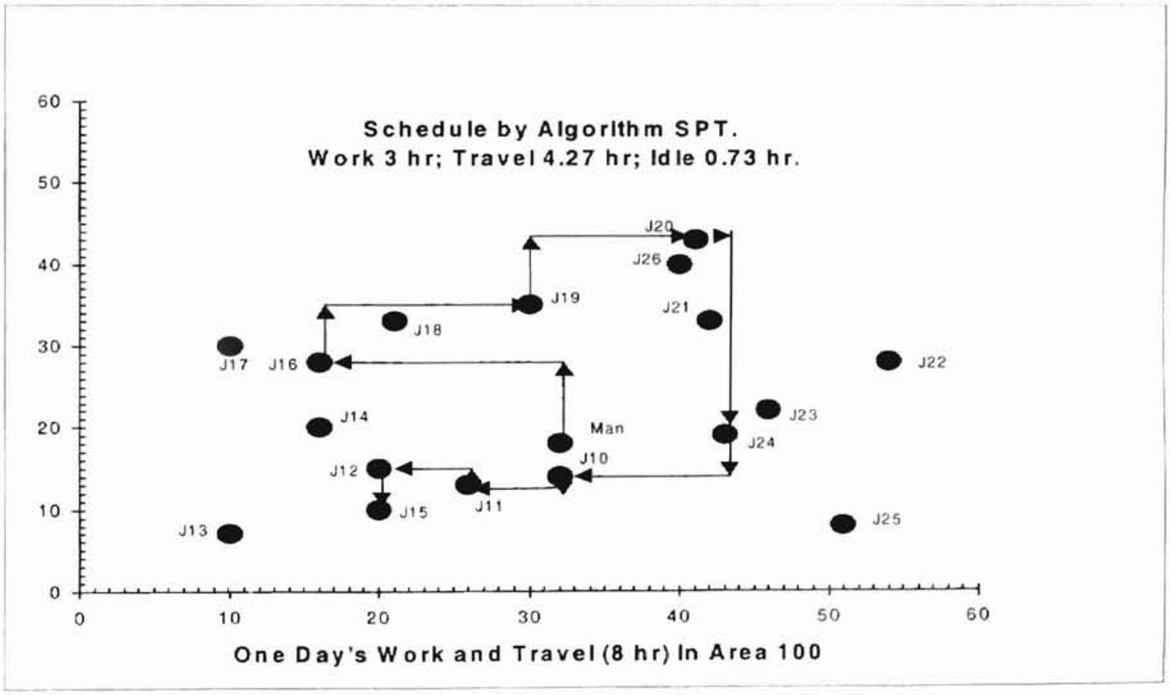

Fig. 20 Possible Travel Route for Man 111 By SPT

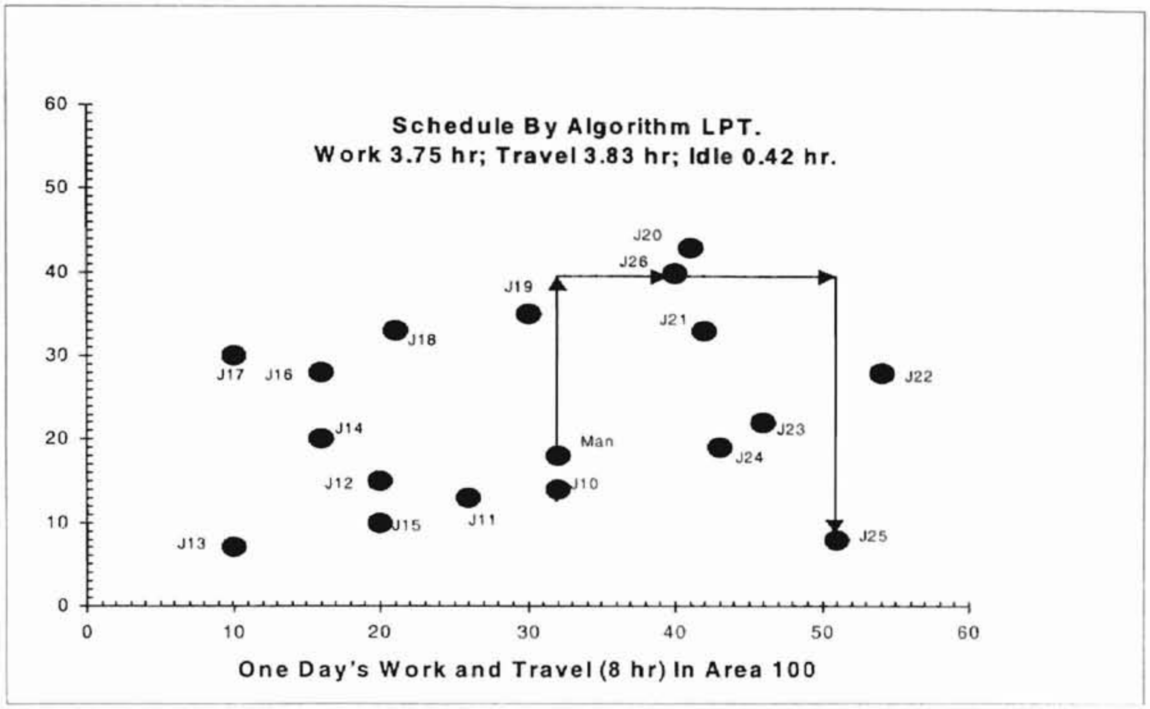

Fig. 21 Possible Travel Route for Man 111 By LPT

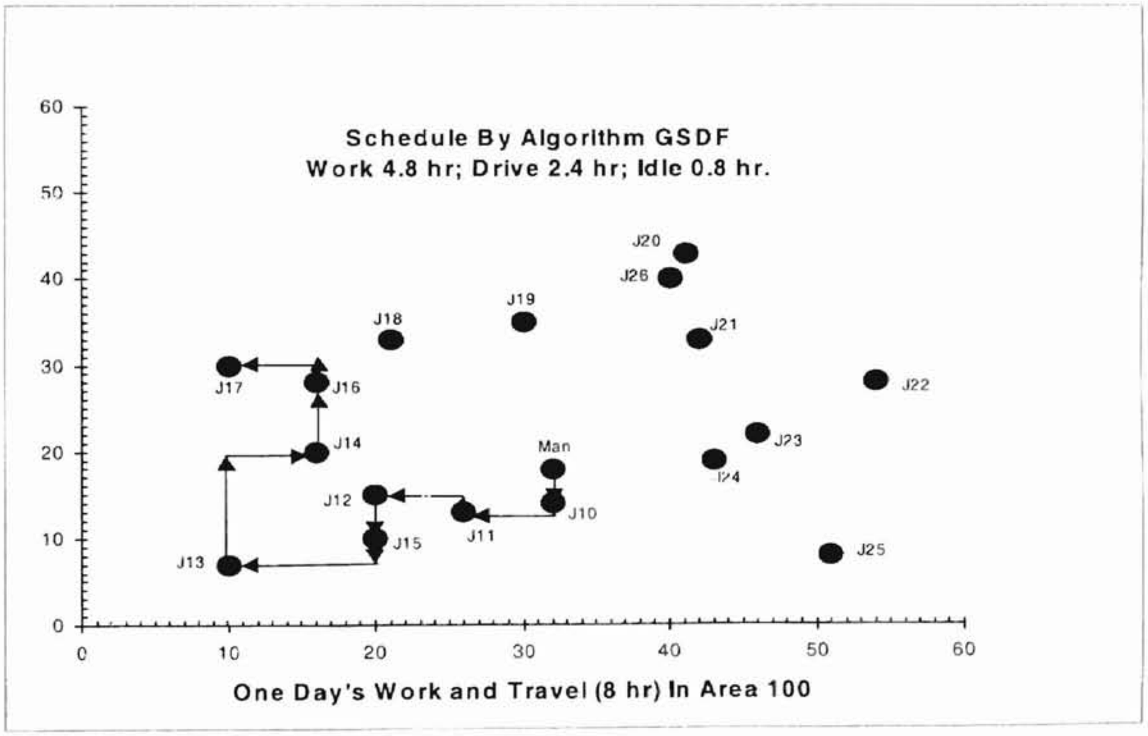

Fig.22 Possible Travel Route for Man 111 By GSDF

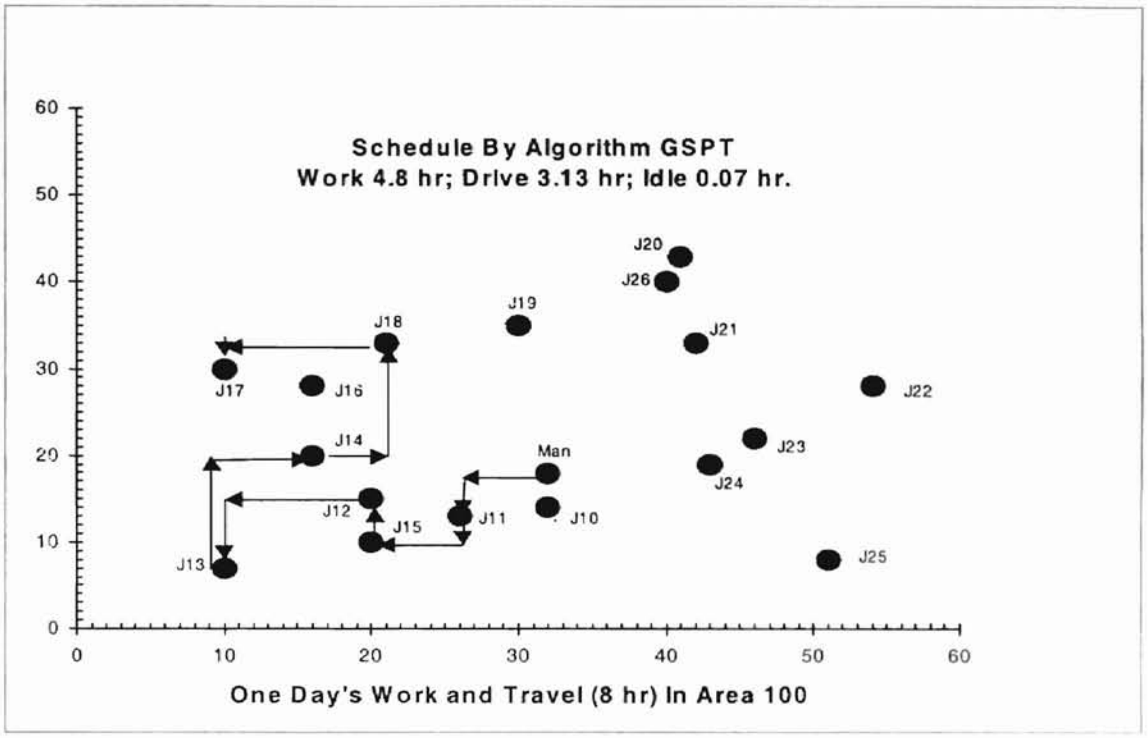

Fig. 23 Possible Travel Route for Man 111 By GSPT

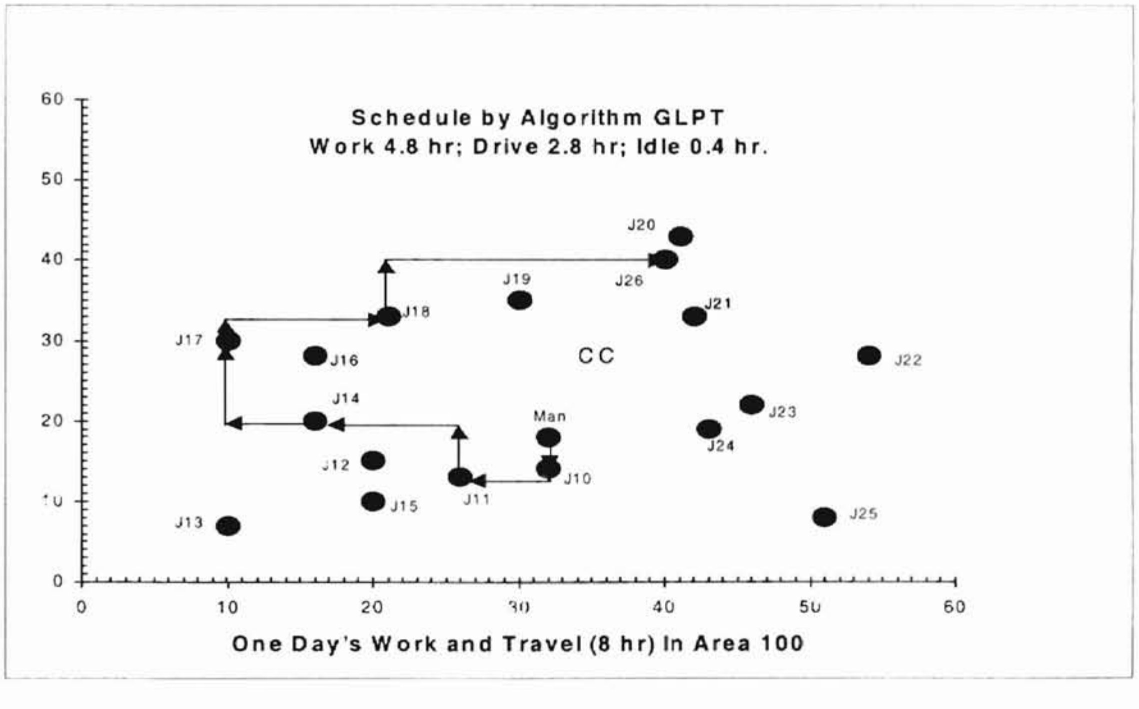

Fig. 24 Possible Travel Route For Man111 By GLPT

2) After eight requests and assignments, the final applied travel routine (see Fig. 25) is none of the above six routines. The result is the combinatorial schedule automatically created by the system. The system chooses one job from the best schedule of the six each time when a technician requested a new job. Thus, the combinatorial schedule is formed. The final schedule is #10, #11, #12, #14, #16, #18, #21, #26. Among these jobs, #10, #11, #12, #16, #26 are chosen from the queue created by SDF, #18 from the queue created by GSDF, #14, and #21 from the queue created by GLPT. The work time is 5.25 hours, travel time is 2.53 hours, idle time is 0.22 hour. For a comparison bar chart see Fig. 26

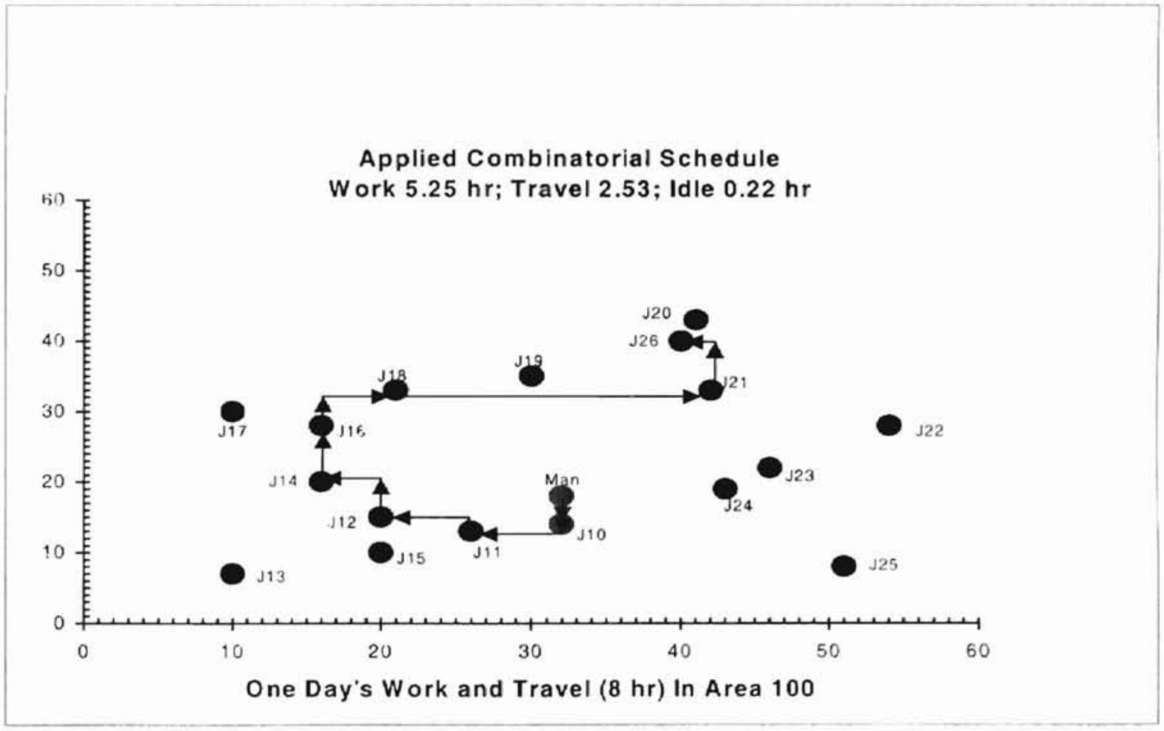

Fig. 25 Final Applied Travel Route for Man 111

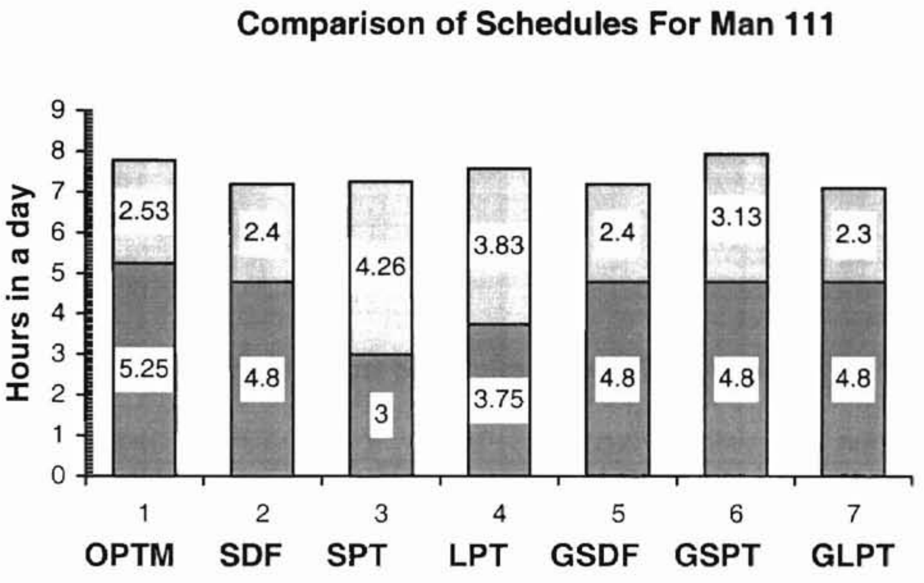

Fig. 26 Bar Chart of Total Work Time and Travel Time

Note: OPTM ---- Final Combinatorial Schedule.

SDF ---- Shortest Travel Distance First Schedule.

SPT ---- Shortest Process Time First Schedule.

LPT ---- Longest Process Time first Schedule.

GSDF ---- Genetic SDF Schedule.

GSPT ---- Genetic SPT Schedule.

GLPT ---- Genetic LPT Schedule.

Observing the above travel routes and the bar chart of schedule comparison, it is clear that a genetic algorithm plays a very important rule in optimizing schedules. In most cases, the final combinatorial schedule intelligently chooses a job from the queue optimized by a genetic algorithm although priority has not been given to these queues.

3.3.2.2 An Eight Hour Day At Normal Jobs Of Long Duration

The typical day's work at normal jobs of long work time tests the performance of algorithm LPT. As expected, the final combinatorial schedule (OPTM) intelligently created a very good solution. By chance, the schedule created by OPTM and genetic LPT (GLPT) are the same. These six schedules are listed in Table 4. Comparison of work time and travel time see Fig 27.

Table 4. Eight-Hours Schedules for M222 In Area 200

Algorithm	Schedule	Work Time (hr)	Travel Time (hr)	No. of Jobs
SDF	#32,#35,#34, #36,#37, #38,#39,#40,#41	4.0	3.23	9
SPT	#31, #33, #35, #37, #38, #40, #43, #32	2.25	4.83	8
LPT	#47,#45,#42	4.75	1.77	3
GSDF	#45,#46,#47,#42	5.25	2.57	84
GSPT	#32,#35,#34,#41,#42,#45	4.75	3.13	6
GLPT	#47, #46, #45, #44, #42	5.75	2.03	5
OPTM	#47, #46, #45, #44, #42	5.75	2.03	5

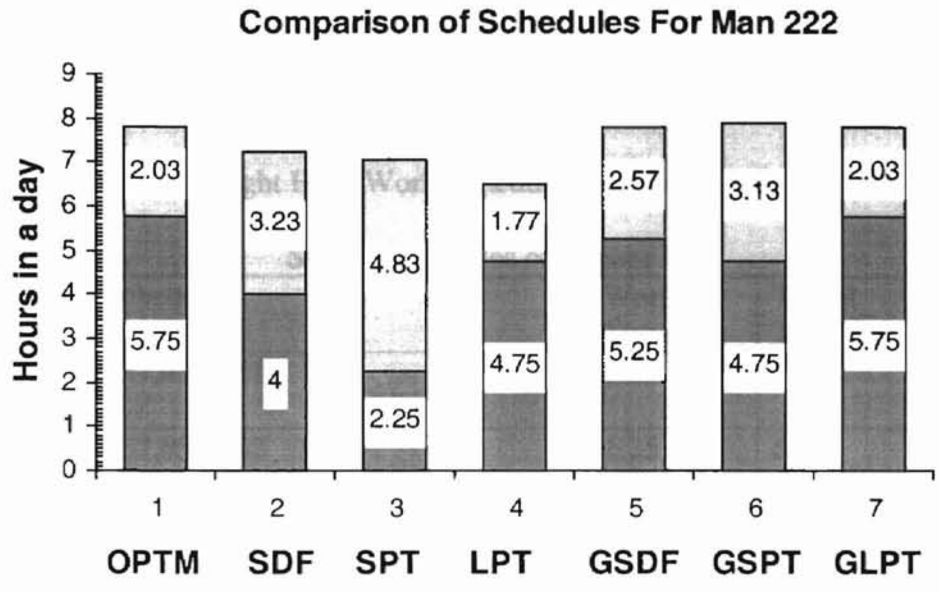

Fig. 27 Comparison of Work Time and Travel Time of SchedulesFor Man 222

3.3.2.3 An Eight Hour Day At Normal And Urgent Jobs In Different Areas

In the following schedule, jobs #51, #52 and #57 are urgent jobs. The system schedules all urgent jobs first. See the Stage I Schedules of Urgent Job table for details. The OPTM schedule is the same as the SDF schedule. Thus, the total time for urgent jobs is 5.05 hours. When all urgent jobs are done, the technician still has 2.95 hours to work for the day. The system automatically turns to normal jobs. See the Stage II Schedule of Normal Job table. The normal jobs are scheduled in accordance with the remaining time of the technician. For work time comparison of schedules, see Figure 28, and for the final combinatorial travel route see Figure 29.

Table 5. Eight Hour Work Schedule For Man 111 In Area 300

Algorithm	Schedule	Work Time (hr)	Travel Time (hr)	No. of Jobs
Stage I Schedules of Urgent Jobs				
SDF	#57,#52,#51	1.25	3.8	3
SPT	#57,#51,#52	1.25	6.5	3
LPT	#51,#52,#57	1.25	4.5	3
GSDF	#57,#52,#51	1.25	3.8	3
GSPT	#57,#51,#52	1.25	6.5	3
GLPT	#51,#52,#57	1.25	4.5	3
OPTM	#57,#52,#51	1.25	3.8	3
Stage II Schedules of Normal Jobs				
SDF	#66,#59,#61	0.75	1.4	3
SPT	#55	0.25	2.2	1
LPT	#63	1.0	1.7	1
GSDF	#60,#65	1.5	1.13	2
GSPT	#60,#65	1.5	1.13	2
GLPT	#60,#65	1.5	1.13	2
OPTM	#60,#65	1.5	1.13	2

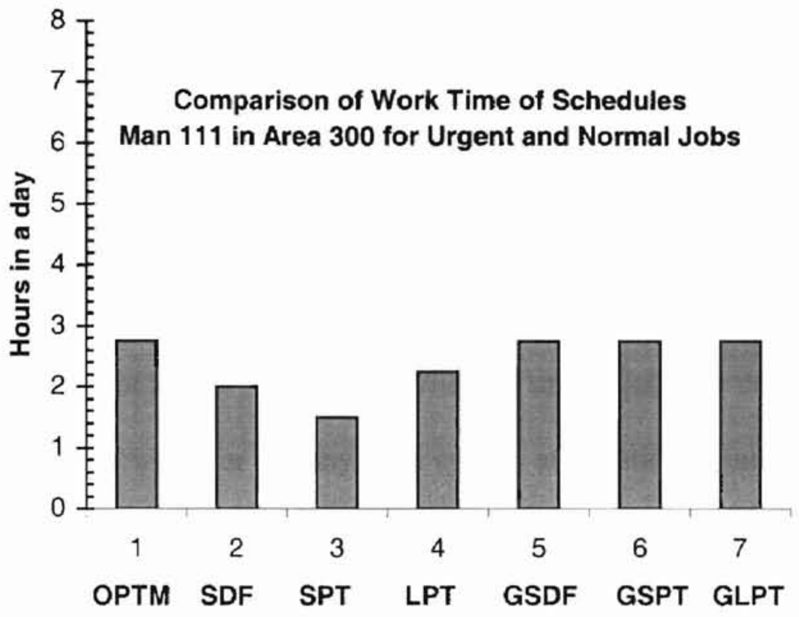

Fig. 28 Comparison Of Work Time for Man 111 in Area 300

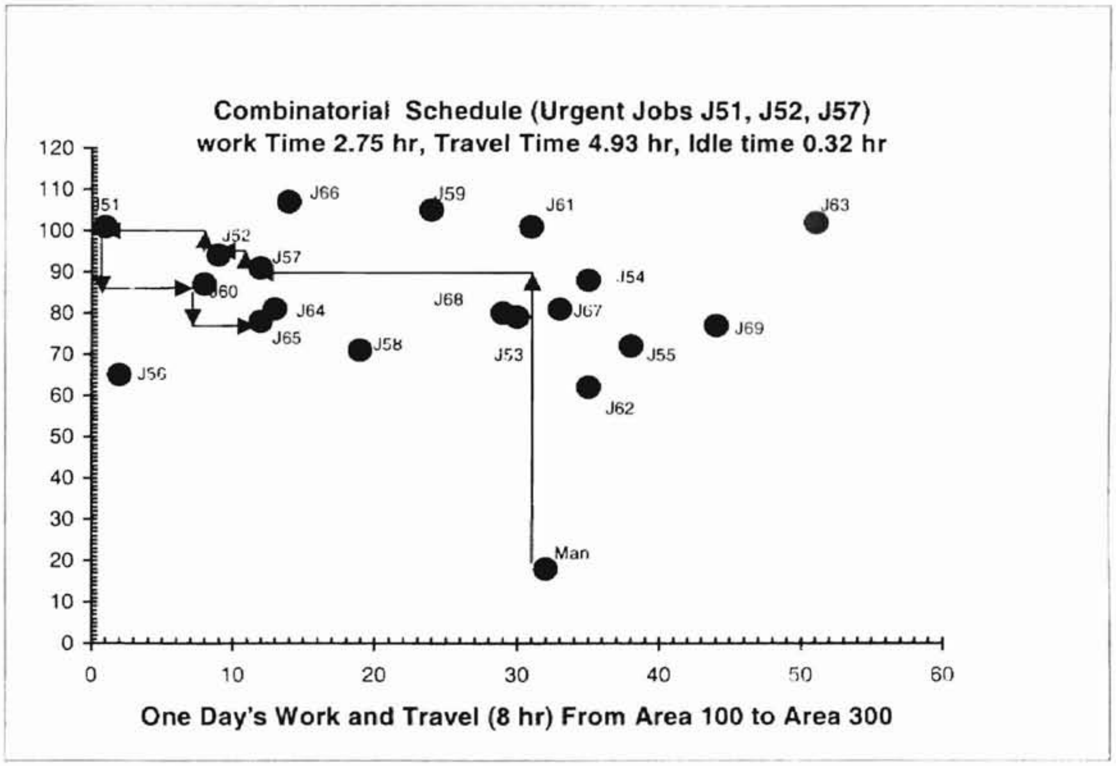

Fig. 29 Travel Route Of Man 111 From Area 100 To Area 300

3.3.2.4 An Eight Hour Day At Normal And Urgent Jobs In Area 400

In the following schedule, jobs #71, #72, #81 and #82 are urgent jobs. The system, as designed, schedules all urgent jobs first. See the Stage I Schedules of Urgent Job table for detail. The OPTM schedule is the same as the SDF schedule. Thus, the total time for urgent jobs is 5.05 hours. When all urgent jobs are done, the technician still has 2.95 hours to work for the day. The system automatically turns to normal jobs. See the Stage II Schedule of Normal Job table. The normal jobs are scheduled in accordance with the remaining time of the technician. For work time comparison of schedules, see Figure 30. For the final combinatorial travel route see Figure 31.

Table 6. Eight-Hour Work Schedule For Man 444 In Area 400

Stage I Schedules of Urgent Jobs

Algorithm	Schedule	Work Time (hr)	Travel Time (hr)	No. of Jobs
SDF	#82,#72,#81,#71	1.37	2.1	4
SPT	#81,#82,#72,#71	1.37	2.63	4
LPT	#71,#72,#82,#81	1.37	3.37	4
GSDF	#82,#72,#81,#71	1.37	2.1	4
GSPT	#81,#82,#72,#71	1.37	2.63	4
GLPT	#71,#72,#82,#81	1.37	3.37	4
OPTM	#82,#72,#81,#71	1.37	2.1	4

Stage II Schedules of Normal Jobs

Algorithm	Schedule	Work Time (hr)	Travel Time (hr)	No. of Jobs
SDF	#88, #87, #80, #77, #75	2.25	2.17	5
SPT	#77, #86, #79	0.8	3.5	3
LPT	#85	0.85	2.2	1
GSDF	#88, #87, #80, #77, #75	2.25	2.17	5
GSPT	#88, #87, #77, #80, #75	2.25	2.17	5
GLPT	#88, #87, #77, #80, #75	2.25	2.17	5
OPTM	#88, #87, #80, #77, #75	2.25	2.17	5

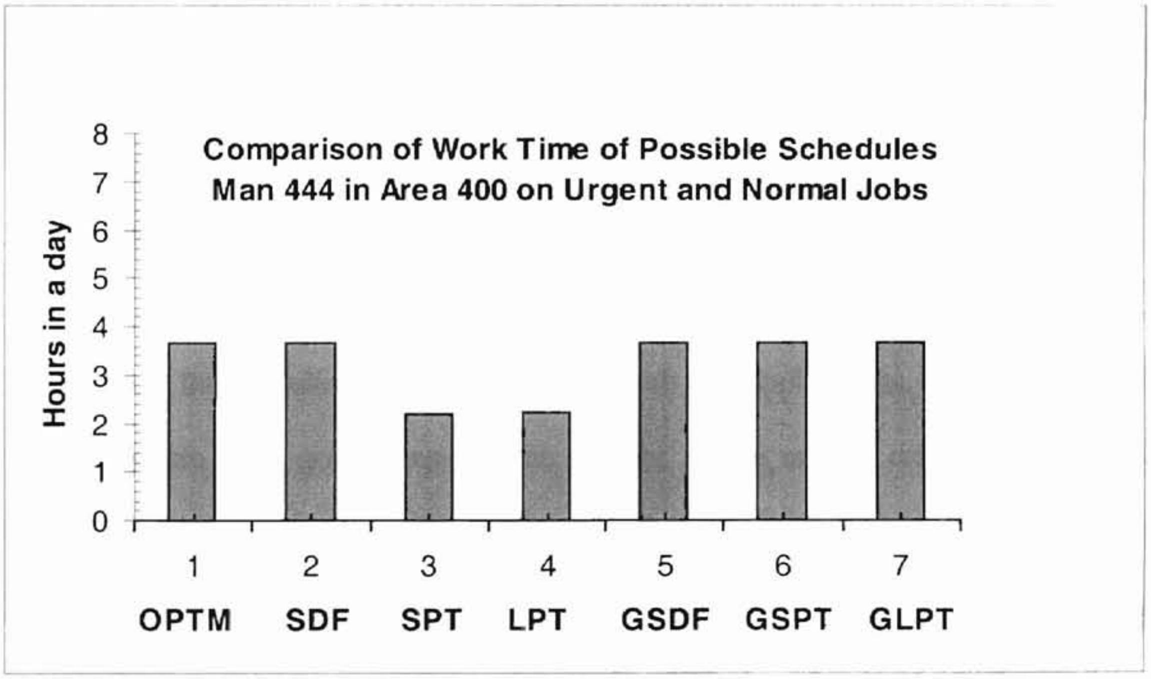

Fig. 30 Comparison Of Work Time for Man 444 in Area 400

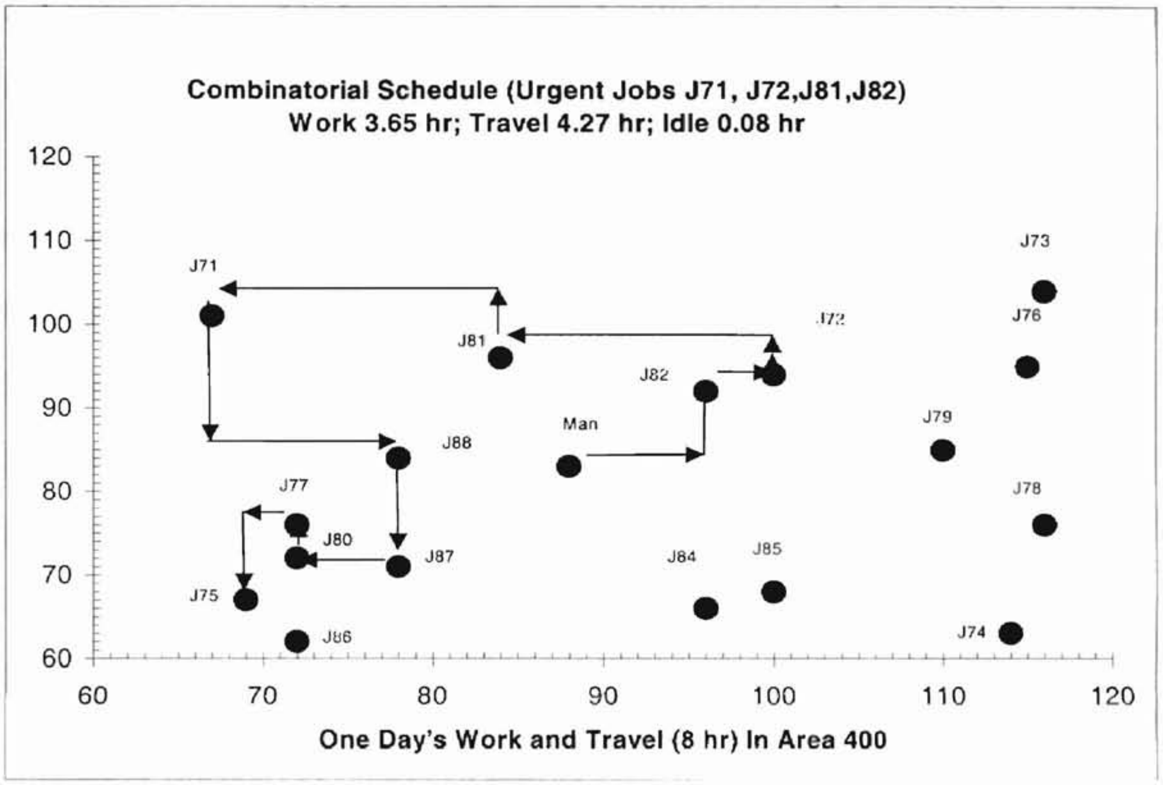

Fig. 31 Travel Route Of Man 444 In Area 400

3.3.3 Observation On The Empirical Application

From the preceding tables of schedules with four applications, we can see that the SDF algorithm only gives consideration to jobs in the nearest distance. Reasonable application of this algorithm will guarantee the shortest travel distance of a day's work, but not the maximum work time of the day. It occasionally schedules a technician with longer idle time or a considerable travel time. The SPT algorithm increases the number of jobs in each schedule. It often schedules a technician with considerable travel distance and less actual work time. The LPT algorithm helps a technician to find jobs with the longest work hours. In turn, this algorithm often results in a situation where jobs with less work time are ignored in favor of jobs with longer duration; the final total work time doesn't increase much, but the travel time does.

The genetic algorithm has provided an unexpected contribution to the success of this combinatorial solution. It keeps a good schedule and continues to find a better chromosome to perfect the schedule. We can see the work time of schedules by GSDF, GSPT and GLPT is longer than that of SDF, SPT, and LPT. The travel time of queues by GSDF, GSPT, and GLPT is shorter than that of SDF, SPT, and LPT. In most cases, the final combinatorial schedule intelligently recognizes GSDF, GSPT, and GLPT. It chooses a job from these queues to create the final best schedule each time a technician requests a job, though priority has not been given to these queues.

CHAPTER 4

SUMMARY AND CONCLUSION

The theory of scheduling is characterized by a virtually unlimited number of problem types. Most research has traditionally been concentrated on deterministic machine scheduling. This thesis emphasizes algorithms for scheduling non-interruptible tasks. Academics require algorithms to be theory-based and written in a mathematical fashion. In the real world the people who need them ask for plain explanation and simple examples. This thesis bridges the gap in communication. Different from other academic theses, this paper keeps the academic style of algorithms, explains them in basic language, and shows simple examples for each algorithm. A total of six types of problems and nineteen algorithms are covered in Chapter 2 to complete this survey.

As the world is accepting scheduling methods, people gradually find that most methods only give consideration to the dedicated conditions. The real world is too complicated. We need to simplify the real applied conditions and combine all possible algorithms to work out an solution. In Chapter 3, such an application is presented. The combinatorial solution successfully uses genetic algorithms to optimize approximately a schedule for the Shortest Process Time First (SPT) algorithm, Longest Process Time First (LPT) algorithm and Shortest Travel Distance First (SDF), and finally selects a job from the best queue of these six. The logic of this empirical application can be summarized in one sentence: simplify the complicated, and optimize approximately the obtained. The advantage of this combinatorial method (OPTM) is that it always provides a very good solution according to the given conditions. The disadvantage is that genetic algorithms need a few seconds to do the calculation. As the practical job scheduling is carried out by the mobile laptop of the technician, the calculation has no influence on the CDS. Since the laptop is on most of the time, it doesn't bother the technician if it needs one extra second for calculation. So this OPTM is successful and applicable.

However, nothing is perfect in the real world. The OPTM can be further improved by building in penalty factors for the work time of each job selected, the number of jobs in each queue, the travel mileage, and the idle time. This work needs the cooperation of the user's financial advisors, production managers and experienced technicians.

BIBLIOGRAPHY

Akers, S. B. [1956] A graphical approach to production scheduling problems. *Operations Research* **4**, 244-245.

Burns, R.N. [1976] Scheduling to minimize the weighted sum of completion times with secondary criteria. *Naval Research Logistics Quarterly* **23**, 125-129.

Baker, K. R. [1974] *Introduction to Sequence and Scheduling*. John Wiley, New York.

Baker, K. R. and Scudder, G. D.[1990] Sequencing with earliness and tardiness penalties: a review. *Operations Research* **38**, 22-36.

Bruno, J. L, Coffman, Jr., E. G. and Stehi, R. [1974] Scheduling independent tasks to reduce mean finishing time. *Communication of the Association of Computing Machinery* **17**, 382-387.

Bruno, J. L. and Gonzalez, T. [1976] *Scheduling Independent Tasks with Release Dates and Due Dates on Parallel Machines*. Technical report 213, Computer Science Department, Pennsylvania State University.

Cheng, T. C. E. and Gupta, M. C. [1989] Survey of scheduling research involving due date determination decision. *European Journal of Operation Research* **38,** 156-166.

Conway, R.W., Maxwell, W.L. and Miller, L.W [1967] *Theory of Scheduling*. Addison-Wesley, Reading, Mass.

Coffman, Jr., E.G., Garey, M.R. and Johnson, D.S. [1978] An application of bin packing to multiprocessor scheduling. *SIAM Journal of Computing* **7**, 1-17.

Davis, E. and Jaffe, J.M. [1981] Algorithms for scheduling tasks on unrelated processors. *Journal of Association of Computing Machinery* **28**, 721-736.

Brown, D. E. and Scherer, W. T. [1995] *Intelligent Scheduling Systems*, Kluwer Academic.

Du, J. and Leung, J.Y.-T. [1988a] Scheduling tree structure tasks with restricted execution times. *Inform. Process. Lett.* 28, 183-188.

Du, J. and Leung, J. Y. –T. [1988b] *Minimizing Mean Flow time with Release Time and Deadline Constraints*. Technical Report, Computer Science Program, University of Texas, Dallas.

Elmaghraby, S. E. and Park, S. H. [1974] Scheduling jobs on a number of identical machines. *AIIE Trans.* **6**, 1-12.

Edmonds, J. [1965d] The Chinese postman's problem (abstract). *Operations Research* **13**, Suppl. 1, B73.

Euler, L. [1736] Solutio problematis ad geometrian situs pertinentis. *Commentarii Academiae Petropolitanae* **8**, 128-140.

French, S. [1982] *Sequencing and Scheduling*, Ellis Horwood

Friesen, D. K. and Langston, M. A. [1986] Evaluation of a MULTIFIT-based scheduling algorithm. *J. Algorithms* **7**, 35-39.

Friesen, D. K. [1987] Tighter bounds for LPT scheduling on uniform processors. *SIAM Journal of Computing* **16**, 554-560.

Friesen, D. K. [1984] Tighter bounds for the multifit processor scheduling. *SIAM*

Journal of Computing **13**, 170-181.

Frederickson, G. N., Hecht, M. S. and Kim, C. E. [1978] Approximation algorithms for some routing problems. *SIAM Journal of Computing* **7**, 178-193.

Graham, R. L. [1978] Combinatorial Scheduling Theory. L.A. Steen (ed.), *Mathematics Today*. Springer-Verlag, New York, 183-211.

Graham, R. L., Lawler, E. L., Lenstra, J. K. and Rinnooy Kan, A. H. G. [1979] Optimization and approximation in deterministic sequencing and scheduling: a survey. *Ann. Discrete. Math.* **5**, 287-326.

Garey, M.R., Johnson, D.S., and Sethi, R. [1976] The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research* **1**, 117-129.

Gonzalez, T. and Sahni, S. [1976] Open shop scheduling to minimize finish time. *Journal of Association of Computing Machinery* **25**, 92-101.

Goldberg, D. E. [1989] *Genetic Algorithms in Search, Optimization, and Machine Learning,* Addison-Wesley.

Herrmann, J., Lee, C.-Y. and Snowdon, J. [1993] A classification of static scheduling Problems," in Complexity in Numerical Optimization, P.M. Pardalos (ed.), pp. 203-253, *World Scientific*.

Hall, L.A.and Shmoys, D.B. [1992] Jackson's rule for single-machine scheduling: making a good heuristic better. *Mathematics of Operations Research* **17**, 22-35.

Haupt, R. [1989] A survey of priority rule-based scheduling. *OR Specktrum*, **11**, 3-16.

Hu, T. C. [1961] Parallel sequencing and assembly line problems. *Operations Research*, **9**, 841-848

Jackson, J.R. [1955] *Scheduling a Production Line to Minimize Maximum Tardiness.*

Research Report 43, Management Science Research Project, University of California, Los Angles.

Johnson, S. M. [1954] Optimal two- and three-stage production schedules with set up time included. *Naval Research Logistics Quarterly* **1**, 61-68.

Kise, H., Ibaraki, T. and Mine, H. [1978] A solvable case of the one machine scheduling problem with ready and due times. *Operations Research* **26**, 121-126.

Kunde, M. and Steppat, H. [1985] First fit decreasing scheduling on uniform multiprocessors. *Discrete Applied Mathematics* **10**, 165-177

Lawler, E. L. [1973] Optimal sequencing of a single machine subject to precedence constraints. *Management Science* **19**, 544-546.

Lawler, E.L., Lenstra, J.K. and Rinnooy Kan, A.H.G. [1982] Recent development in deterministic sequencing and scheduling: a survey, in *Deterministic and Stochastic Scheduling*, Dempster, Lenstra & Rinnooy Kan [1982] 35-73.

Lenstra, J. K. [1977] *Sequencing by Enumerative Methods*. Mathematical Centre Tracts 69, Centre for Mathematics and Computer Science, Amsterdam.

Moore, J. M. [1968] an n-job, one machine sequencing algorithm for minimizing the number of late jobs. *Management Science* **15**, 102-109.

Morrison, J. F. [1988] A note on LPT scheduling. *Operations Research Letters*, **7**, 77-79.

Pinedo, Michael [1995] *Scheduling Theory, Algorithms, and Systems*. Prentice Hall.

Nowicki, E. and Smutnicki, C [1987] On lower bound on the minimum maximum lateness on one machine subject to release date. *Opsearch* **24**, 106-110.

Panwalker, S. S. and Woollam, C. R. [1979] Flow-shop scheduling problem with no in-processing waiting: a special case. *J. Opl. Res. Soc.* **30**, 661-664.

Panwalker, S. S. and Woollam, C. R. [1980] Ordered flow-shop problems with no in-processing waiting: further results. *J. Opl. Res. Soc.* **31**, 1039-1043.

Panwalker, S. S. and Iskander, W. [1977] A survey of scheduling rules. *Operations Research* **25**, 45-61.

Parker, R. G. [1995] *Deterministic Scheduling Theory*. Chapman & Hall.

Cretienne, P., Coffman Jr., E. G., Lenstra, J. K. and Liu, Zhen [1995] *Scheduling Theory and Its Applications*, John Wiley & Sons.

Brucker, Peter [1998] *Scheduling Algorithms*, Springer-Verlag Berlin. Heidelberg.

Rinnooy Kan, A.H.G. [1976] *Machine Scheduling Problems: Classification, Complexity and Computations*. Mutinus Nijhoff, The Hague, Holland.

Robert E. D. Woolsey and Huntington S. Swanson [1969] *Operatiosn Research for Immediate Application: A Quick & Dirty Manual*. Harper & Row.

Roger Kerr and Elizabeth Szelke[1995] *Artificial Intelligence in Reactive Scheduling*, Chapman & Hall.

Silberschatz, A.and Galvin, P. B. [1998] *Operating System Concepts*, Addison-Wesley.

Sturm, L. B. J. M. [1970] A simple optimality proof of Moore's sequencing algorithm. *Management Science* **17**, B116-B118.

Smith, W.E. [1956] Various optimizers for single state production. *Naval Research Logistics Quarterly* **3**, 59-66.

Swarzc, W. [1977] Optimal two machine orderings in the 3 x n flow-shop problem. *Operations Research* **25**, 70-77.

Smith, M. L., Panwalker, S. S. and Dudek, R. A. [1976] flow-shop sequencing problem with ordered processing time matrices: a general case. *Naval Research Logistics*

Quarterly **22**, 481-486.

Sahni, S. [1976] Algorithms for scheduling independent tasks. *Journal of Association for Computing Machinery* **23**, 116-127.

Sahni, S. and Cho, Y. [1980] Scheduling independent tasks with due times on a uniform processor system. *Journal of Association for Computing Machinery* **27**, 550-563.

Sturm, L. B. J. M. [1970] A simple optimality proof of Moore's sequencing algorithm. *Management Science* **17**, B116-B118.

Tanaev, V. S., Gordon, V. S. and Shafransky, Y. M. [1994] *Scheduling Theory. Single-State Systems*, Kluwer Academic.

Tanaev, V. S., Gordon, V. S. and Shafransky, Y. M. [1994] *Scheduling Theory. Multi-State Systems*, Kluwer Academic.

Thomas M. Cook and Robert A. Russell [1981] *Introduction to Management Science*, Prentice-Hall.

Van Wassenhov, L.N. and Gelders, L.F. [1980] Solving a bicriterion-scheduling problem. *European Journal of Operation Research* **4**, 42-48.

Van Wassenhov, L.N. and Baker, K. R. [1980] A bicriterion approach to time/cost trade-offs in sequencing. Paper presented at the 4th European Congress on Operational Research, Cambridge, England, July 22-25, 1980. Submitted to A.I.I.E. Trans.

White, D. J. [1969] *Dynamic Programming*, Oliver and Boyd, Edinburgh.

Yueh Ming-I [1976] On the n job, m machines sequencing problem of a flow-shop. In Operation Research 1975, Haley, K. B. (Ed.), North Holland, Amsterdam.

APPENDIX A

C PROGRAMMING CODE FOR COMBINATORIAL SCHEDULING NON-INTERRUPTIBLE TASKS

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
#include<time.h>

#define TRUE 0
#define FALSE 1
#define EXSIZE 256              //array size
#define FULL 81                 //array size
#define HALF 40                 //array size
#define QUART 20                //array size
#define JOBHD 1                 //a flag for job queue
#define MANHD 2                 //a flag for technician queue
#define SPEED 30                //driving speed
#define REPRODUCTION 40320      //reproduction number

typedef struct TheJob{          //node for a job
      int JobID;                //job identification number
      int ZipCode;              //zip code
      char Type[QUART];         //job type
      char Status[QUART];       //job status
      float EWkTime;            //estimated work time
      float AWkTime;            //actual work time
      char DueDay[QUART];       //job due day
      char StTime[QUART];       //job start time
      char FinTime[QUART];      //job finish time
      char StDate[QUART];       //job start date
      char FinDate[QUART];      //job finish date
      char Address[FULL];       //job location
      int LocX;                 //job coordinate X
      int LocY;                 //job coordinate Y
      float Mileage;            //driving mileage
      struct TheJob *next;
}TheJob;

typedef struct TheMan{          //node for a technician
```

```
      int ManID;                //technician identification number
      int ZipCode;              //zip code
      char Name[QUART];         //name of technician
      char Address[FULL];       //technician home address
      char LoginTime[HALF];     //technician log in time
      char LogoutTime[HALF];    //technician log out time
      int LocX;                 //technician start coordinate X
      int LocY;                 //technician start coordinate Y
      float TwkTime;            //work time
      float Mileage;            //driving mileage
      struct TheMan *next;
}TheMan;

TheJob* JobHead=NULL;           //head of job queue
TheJob* NewHead=NULL;           //temp pointer
TheMan* ManHead=NULL;           //head of technician queue
TheJob* UrgHead=NULL;           //head of urgent job queue
TheJob* NomHead=NULL;           //head of normal job queue

//functions used in this program

int AbsJulian(char*);     //calculate Julian day numbers
int manageInput();        //filter out input like empty file, tab, space
void LinkJobs(TheJob*); //build job queue
void InputJob();          //put input job data to job node
void PrintJobs(TheJob*);//print input jobs
void LinkMan(TheMan*);  //build technician queue
void InputMan();          //put input technician data to technician node
void PrintMan();          //print technician login information
void SortDueDay(int);   //sort jobs by due day
int FindDueDay();         //find the due day of a job
void SetQueue(int*, int);       //set a temp queue
void LinkMe(TheJob* );          //insert a job node into linked list
void DeleteMe(TheJob* );        //delete a job node from linked list
int TimeDuration(char*, char*, char*, char*);//calculate work duration
void JobDone(int, char*, char*);            //mark job done
void ArriveOnSite(int, char*, char*);//mark technician arrives on site
void LogoutAccept(int);         //check if log out is accepted
int IsNameinList(int);          //check employee list
int LoginAccept(int);           //check if log in is accepted
void ScheduleList(float, int);        //build urgent and normal job lists
void FindUrgentJobs(TheJob*, float*, float, int);//pick out urgent jobs
void FindNormalJobs(TheJob*, float*, float, int);//pick out normal jobs
void NewJobList(TheJob* );      //To build urgent and normal job queues
int Scheduling(int*, float, int);     //manage overall scheduling
int ScheduleJobs(TheJob*, int*, float, int);     //do actual scheduling
void SetStartLocation(int, int);                 //reset start location
int SPTqueue(int*, float, int, int, int*);//by Shortest time first
int LPTqueue(int*, float, int, int);       //by longest time first
int SDFqueue(int*, float, int, int);//by shortest distance first
int Genqueue(int*, float, int, int);//by genetic algorithm
int TravelDistance(int, int, int);  //calculate travel distance
int TravelTime(int, int*, int*);    //calculate travel time
void FindJobLocation(int, int*, int*);     //find job location
int FindWkTime(int);                       //find work time
void FindManLocation(int, int*, int*);     //find technician's location
void LinkNewMe(TheJob*);        //building urgent or normal job queue
void SetWkTime(int, int, float);          //record work time of a job
float getWktime(int);                     //fetch work time
void printTheJob(int, int *, int *);      //print job information
void printTheMan(int);              //print technician's work report
void ChangeStatus(TheJob*, int);          //change job status
int IsZipCodeValid(int, int);             //check zip code validation
void printQueue(int*, float,int, int, FILE*);//print work summary


/********************************************************************/
/*     Read input job data and store it to job node                 */
/********************************************************************/
void InputJob()
{
      FILE *fp;          //FILE pointer
      TheJob *JobPtr;
      char T[FULL];      //temp buffer
      char* str;         //character pointer
      int Jobcount=0;    //job counter

      if(( fp = fopen("JobX.dat","r")) == NULL){  //check input file
      printf("Input file could not be opened\n");
            exit(1);
      }
      JobPtr = ((TheJob*)malloc(sizeof (TheJob)));

      while((fgets(T, FULL, fp))!=NULL){         //read input data
            fgets(JobPtr->Address, FULL, fp);    //get job location
            str = strtok(T, "~\n\0");            //get job ID
            JobPtr->JobID = atoi(str);
            str = strtok(NULL, "~\n\0");         //get zip code
            JobPtr->ZipCode = atoi(str);
            str = strtok(NULL, "~\n\0");         //get job type
            strcpy(JobPtr->Type, str);
            str = strtok(NULL, "~\n\0");         //get job status
            strcpy(JobPtr->Status, str);
            str = strtok(NULL, "~\n\0");  //get estimated work time
            JobPtr->EWkTime = atof(str);
            str = strtok(NULL, "~\n\0");         //get job due day
            strcpy(JobPtr->DueDay, str);
            str = strtok(NULL, "~\n\0");         //get coordinate X
            JobPtr->LocX = atoi(str);
            str = strtok(NULL, "~\n\0");         //get coordinate Y
            JobPtr->LocY = atoi(str);
            JobPtr->AWkTime=0;             //initialize actual work time
            JobPtr->Mileage=0;                   //initialize mileage
            strcpy(JobPtr->StTime, "     ");     //initialize start time
            strcpy(JobPtr->FinTime, "     ");//initialize finish time
            strcpy(JobPtr->StDate, "     ");     //initialize start date
            strcpy(JobPtr->FinDate, "     ");//initialize finish date
            JobPtr->next=NULL;
            LinkJobs(JobPtr);              //insert job to linked list
            Jobcount++;                    //count input jobs
            JobPtr = ((TheJob*)malloc(sizeof (TheJob)));
      }
      fclose(fp);
      SortDueDay(Jobcount);   //sort input job by due day
```

```
        JobHead=NewHead;
        NewHead=NULL;
}


/**********************************************************************/
/*      Insert jobs into linked list                                  */
/**********************************************************************/
void LinkJobs(TheJob* Pt)
{
        TheJob* walk=NULL;          //pointer
        walk=JobHead;

        if(JobHead==NULL)
                JobHead=Pt;         //put job to head
        else{
                while(walk->next != NULL)//find the end of queue
                        walk=walk->next;
                walk->next=Pt;
        }

}

/**********************************************************************/
/*      Print job assignment information when it is scheduled to      */
/*      the dedicated technician.                                     */
/**********************************************************************/
void printTheJob(int id, int *x, int *y)
{
        FILE* fp;                       //input file pointer
        int t=0,d=0;
        TheJob* JobPtr;                 //node pointer
        fp=fopen("fout", "a");
        JobPtr=JobHead;

        while(JobPtr->JobID!=id)        //find the scheduled job
                JobPtr=JobPtr->next;

        //get travel distance to the job
        d=abs(*x-(JobPtr->LocX))+abs(*y-(JobPtr->LocY));
        t=60*d/SPEED;

        printf("\nYou are to work on:");
        printf("job #%d.\n", JobPtr->JobID);
        printf("Job type:       %s\n", JobPtr->Type);
        printf("Job location: %s", JobPtr->Address);
        printf("Job due day:  %s\n", JobPtr->DueDay);
        printf("Estimated working time:      %2.2f hours.\n", JobPtr->
        EWkTime);
        printf("Estimated travel distance: %d miles.\n", d);
        printf("Estimated driving time:     %d minutes.\n", t);

        *x=JobPtr->LocX;
        *y=JobPtr->LocY;

        fprintf(fp, "%s", "/*************************************/\n");
        fprintf(fp,"%s", "\nYou are to work on:");
        fprintf(fp,"job #%d.\n", JobPtr->JobID);
        fprintf(fp,"Job type:      %s\n", JobPtr->Type);
        fprintf(fp,"Job location: %s", JobPtr->Address);
        fprintf(fp,"Job due day:  %s\n", JobPtr->DueDay);
        fprintf(fp,"Estimated working time:     %2.2f hours.\n", JobPtr->
        EWkTime);
        fprintf(fp,"Estimated travel distance: %d miles.\n", d);
        fprintf(fp,"Estimated driving time:     %d minutes.\n", t);

        fclose(fp);
}


/**********************************************************************/
/*      Print a technician's work report                              */
/**********************************************************************/
void printTheMan(int id)
{
        FILE* fp;                  //file pointer
        TheMan* ManPtr;
        fp=fopen("fout", "a");
        ManPtr=ManHead;
        while(ManPtr->ManID!=id)        //find the technician
                ManPtr=ManPtr->next;
        printf("Name:              %s\n", ManPtr->Name);
        printf("Employee ID:       %d\n", ManPtr->ManID);
        printf("From:              %s", ManPtr->Address);
        printf("Login time:        %s\n", ManPtr->LoginTime);
        printf("Logout time:       %s\n", ManPtr->LogoutTime);
        printf("Total actual working time:%2.2f hours\n", ManPtr->
        TWkTime/60);
        printf("Total driving mileage:%2.2f miles\n", ManPtr->Mileage);
        printf("Total time for today: %2.2f hours\n", ManPtr->
        TWkTime/60+ManPtr->Mileage/SPEED);

        fprintf(fp,"%s", "            FINAL REPORT\n");
        fprintf(fp, "%s", "/************************************/\n");
        fprintf(fp,"Name:              %s\n", ManPtr->Name);
        fprintf(fp,"Employee ID:       %d\n", ManPtr->ManID);
        fprintf(fp,"From:              %s", ManPtr->Address);
        fprintf(fp,"Login time:        %s\n", ManPtr->LoginTime);
        fprintf(fp,"Logout time:       %s\n", ManPtr->LogoutTime);
        fprintf(fp,"Total working time:%2.2f hours\n", ManPtr->
        TWkTime/60);
        fprintf(fp,"Total driving mileage:%2.2f miles.\n", ManPtr->
        Mileage);
        fprintf(fp,"Total time for today: %2.2f hours\n", ManPtr->
        TWkTime/60+ManPtr->Mileage/SPEED);

        fclose(fp);
}


/**********************************************************************/
/*      Record work time of a technician on a job                     */
/**********************************************************************/
void SetWkTime(int jid, int mid, float tm)
```

```
{
      TheMan* ManPtr;
      TheJob* JobPtr;
      ManPtr=ManHead;          //point to technician queue
      JobPtr=JobHead;          //point to job queue

      while(JobPtr->JobID!=jid)
            JobPtr=JobPtr->next;      //find job node

      tm=JobPtr->EWkTime;            //record work time
      JobPtr->AWkTime=tm;
      tm=tm*60;

      while(ManPtr->ManID!=mid)      //find the technician
            ManPtr=ManPtr->next;

      ManPtr->TWkTime=ManPtr->TWkTime + tm;//sum total work time
}

/*******************************************************************/
/*    Print all input jobs                                         */
/*******************************************************************/
void PrintJobs(TheJob* HD)
{
      FILE* fp;
      TheJob* JobPtr;
      fp=fopen("fout", "a");
      JobPtr=HD;

      if(JobPtr==NULL)
            return;

      while(JobPtr->next!=NULL){
            fprintf(fp,"JobID: %d \t", JobPtr->JobID);
            fprintf(fp,"ZipCode: %d \t", JobPtr->ZipCode);
            fprintf(fp,"Job Type: %s \t", JobPtr->Type);
            fprintf(fp,"Job Status: %s \n", JobPtr->Status);
            fprintf(fp,"Estimated work time: %2.2f \t", JobPtr->
            EWkTime);
            fprintf(fp,"Actual work time: %2.2f \t", JobPtr->AWkTime);
            fprintf(fp,"Job Due Day: %s \n", JobPtr->DueDay);
            fprintf(fp,"Job start time: %s \t", JobPtr->StTime);
            fprintf(fp,"Job complete time time: %s \n", JobPtr->
            FinTime);
            fprintf(fp,"Job start date time: %s \t", JobPtr->StDate);
            fprintf(fp,"Job complete date %s \n", JobPtr->FinDate);
            fprintf(fp,"Locatin X: %d \t", JobPtr->LocX);
            fprintf(fp,"Location Y: %d \n", JobPtr->LocY);
            fprintf(fp,"Address: %s \n", JobPtr->Address);
            JobPtr=JobPtr->next;
      }
      fprintf(fp,"JobID: %d \t", JobPtr->JobID);
      fprintf(fp,"ZipCode: %d \t", JobPtr->ZipCode);
      fprintf(fp,"Job Type: %s \t", JobPtr->Type);
      fprintf(fp,"Job Status: %s \n", JobPtr->Status);
      fprintf(fp,"Estimated work time: %2.2f \t", JobPtr->EWkTime);
      fprintf(fp,"Actual work time: %2.2f \t", JobPtr->AWkTime);
```

```
        fprintf(fp,"Job Due Day: %s \n", JobPtr->DueDay);
        fprintf(fp,"Job start time: %s \t", JobPtr->StTime);
        fprintf(fp,"Job complete time time: %s \n", JobPtr->FinTime);
        fprintf(fp,"Job start date time: %s \t", JobPtr->StDate);
        fprintf(fp,"Job complete date %s \n", JobPtr->FinDate);
        fprintf(fp,"Locatin X: %d \t", JobPtr->LocX);
        fprintf(fp,"Location Y: %d \n", JobPtr->LocY);
        fprintf(fp,"Address: %s \n", JobPtr->Address);
        fclose(fp);
}


/*********************************************************************/
/* read technician information and store it to technician node      */
/*********************************************************************/
void InputMan()
{
        FILE *fp;
        TheMan *ManPtr;
        char T[FULL];      //temp buffer
        char* str;

        if(( fp = fopen("Man.dat","r")) == NULL){//check input file
                printf("Input file could not be opened\n");
                exit(1);
        }

        ManPtr = ((TheMan*)malloc(sizeof (TheMan)));
        fgets(T, FULL, fp);

        while((fgets(ManPtr->Address, FULL, fp))!=NULL){
                str = strtok(T, "~\n\0");        //get technician address
                strcpy(ManPtr->Name, str);       //get technician name
                str = strtok(NULL, "~\n\0");
                ManPtr->ManID = atoi(str);       //get technician ID number
                str = strtok(NULL, "~\n\0");
                ManPtr->ZipCode = atoi(str);     //get technician zip code
                str = strtok(NULL, "~\n\0");
                ManPtr->LocX = atoi(str);        //get technician coordinate X
                str = strtok(NULL, "~\n\0");
                ManPtr->LocY = atoi(str);        //get technician coordinate Y
                strcpy(ManPtr->LoginTime, "     "); //initialize login time
                strcpy(ManPtr->LogoutTime, "     ");//initialize logout time
                ManPtr->TWkTime=0;               //initialize total work time
                ManPtr->Mileage=0;               //initialize driving mileage
                ManPtr->next=NULL;
                LinkMan(ManPtr);           //put technician to linked list
                ManPtr = ((TheMan*)malloc(sizeof (TheMan)));
                fgets(T, FULL, fp);//get the first line to Buffer T;
        }
        fclose(fp);
}


/*********************************************************************/
/*     Insert a technician node to linked list                      */
/*********************************************************************/
```

```
void LinkMan(TheMan* Pt)
{
	TheMan* walk=NULL;
	walk=ManHead;

	if(ManHead==NULL)
		ManHead=Pt;
	else{
		while(walk->next != NULL)	//find the node
			walk=walk->next;
		walk->next=Pt;			//insert it to the list
	}

}


/*****************************************************************************/
/*	Print all technicians who have been registered					*/
/*****************************************************************************/
void PrintMan()
{
	FILE* fp;
	TheMan* ManPtr;
	ManPtr=ManHead;

	fp=fopen("fout", "a");
	fprintf(fp, "%s", "	ALL TECHNICIANS ON THE LIST\n");
	fprintf(fp, "%s", "/**********************************/\n");

	while(ManPtr->next!=NULL){
		fprintf(fp,"Name: %s \t", ManPtr->Name);
		fprintf(fp,"ManID: %d \t", ManPtr->ManID);
		fprintf(fp,"ZipCode: %d \n", ManPtr->ZipCode);
		fprintf(fp,"Login Time: %s \t", ManPtr->LoginTime);
		fprintf(fp,"Logout Time: %s \n", ManPtr->LogoutTime);
		fprintf(fp,"Total work time: %s \n", ManPtr->TWkTime);
		fprintf(fp,"Driving Mileage: %s \n", ManPtr->Mileage);
		fprintf(fp,"LocX: %d \t", ManPtr->LocX);
		fprintf(fp,"LocY: %d \n", ManPtr->LocY);
		fprintf(fp,"Address: %s \n", ManPtr->Address);
		ManPtr=ManPtr->next;
	}
	fprintf(fp,"Name: %s \t", ManPtr->Name);
	fprintf(fp,"ManID: %d \t", ManPtr->ManID);
	fprintf(fp,"ZipCode: %d \n", ManPtr->ZipCode);
	fprintf(fp,"Login Time: %s \t", ManPtr->LoginTime);
	fprintf(fp,"Logout Time: %s \n", ManPtr->LogoutTime);
	fprintf(fp,"Total work time: %s \n", ManPtr->TWkTime);
	fprintf(fp,"Driving Mileage: %s \n", ManPtr->Mileage);
	fprintf(fp,"LocX: %d \t", ManPtr->LocX);
	fprintf(fp,"LocY: %d \n", ManPtr->LocY);
	fprintf(fp,"Address: %s \n", ManPtr->Address);

	fclose(fp);
}

/*****************************************************************************/
```

```
/*    Calculate Julian day numbers                                       */
/************************************************************************/
int AbsJulian(char* MDY)
{
      int M;              //month
      int D;              //day
      int Y;              //year
      int ndim[13];       //number of days in a month
      int Lpyrs;          //leap year
      int days;
      int d=0;
      char* str;
      char DD[QUART];
      ndim[1]=0;          //number of days at the beginning of a month
      ndim[2]=31;         //Feburary
      ndim[3]=59;         //March
      ndim[4]=90;         //April
      ndim[5]=120;        //May
      ndim[6]=151;        //June
      ndim[7]=181;        //July
      ndim[8]=212;        //August
      ndim[9]=243;        //September
      ndim[10]=273;       //October
      ndim[11]=304;       //November
      ndim[12]=334;       //December

      strcpy(DD, MDY);
      str=strtok(DD,"/\0\n");
      M=atoi(str);                    //get month number
      str=strtok(NULL,"/\0\n");
      D=atoi(str);                    //get day number
      str=strtok(NULL,"/\0\n");
      Y=atoi(str);                    //get year number

      if(Y >= 2000)
            Y=(Y-2000)+100;
      else{
            if(Y>1900)
                  Y=Y-1900;
      }

      if(Y<95)
            Y=Y+100;

      Lpyrs=(Y/4);

      if(M<=2)
            D=D-1;

      days=(Y/2)*365 + (Y/2)*365 + Lpyrs + ndim[M] + D;

      return days;
}

/************************************************************************/
/*    Sort input jobs in job queue according to their due days          */
/************************************************************************/
```

```
void SortDueDay(int num)
{
	TheJob *walk, *Ptr, *Head=NULL;
	int queue[EXSIZE];	//job queue
	int i=0;

	//put job ID in queue to represent the job, sort jobs in due day
	SetQueue(queue, num);

	while(num!=0){//put jobs in linked list in sorted order
		walk=Ptr=JobHead;
		while(walk->JobID!=queue[i])
			walk=walk->next;

		LinkMe(walk);
		DeleteMe(walk);
		i++;
		num--;
	}

	JobHead=NewHead;
}


/*******************************************************************/
/*	Insert a job node into new scheduled linked list	*/
/*******************************************************************/
void LinkMe(TheJob* Pt)
{
	TheJob* walk=NULL;
	walk=NewHead;

	if(walk==NULL)
		NewHead=Pt;
	else{
		while(walk->next!=NULL)
			walk=walk->next;
		walk->next=Pt;
	}
}


/*******************************************************************/
/*	Delete a job node from linked list	*/
/*******************************************************************/
void DeleteMe(TheJob* Pt)
{
	TheJob* walk=NULL;
	walk=JobHead;

	if(Pt==JobHead&&Pt->next==NULL)
		JobHead=NULL;

	else if(Pt==JobHead&&Pt->next!=NULL){
		JobHead=Pt->next;
		Pt->next=NULL;
	}
```

```
        else{
                while(walk->next!=Pt)//find the job
                        walk=walk->next;

                walk->next=Pt->next;
                Pt->next=NULL;
        }
}

/***********************************************************************/
/*      Rebuild temporary queue according to due day                   */
/***********************************************************************/
void SetQueue(int* Q, int cnt)
{
        TheJob *walk;
        int dueday=0;           //due day
        int i=0, j=0;
        int id=0;               //job ID
        walk=JobHead;

        //put JobID to queue array
        while(walk->next!=NULL){
                Q[i++]=walk->JobID;
                walk=walk->next;
        }
        Q[i++]=walk->JobID;

        for(i=1; i<cnt; i++){
                dueday=FindDueDay(Q[i]);        //find job due day
                id=Q[i];
                for(j=i; j>0 && FindDueDay(Q[j-1])>dueday; j--)
                        Q[j]=Q[j-1];            //sort job queue by due day

                Q[j]=id;
        }
}


/ *********************************************************************/
/*  Find job due day from list, return due day in Julian day number  */
/***********************************************************************/
int FindDueDay(int JID)
{
        TheJob *walk;
        int dueday=0;           //job due day
        walk=JobHead;

        while(walk->JobID!=JID)
                walk=walk->next;                //find the job

        dueday=AbsJulian(walk->DueDay);         //get Julian day number

        return dueday;
}

/***********************************************************************/
```

```
/*    Collect jobs from job list for today and call to build urgent */
/*    job queue and normal job queue.                               */
/*******************************************************************/
void ScheduleList(float hr, int Zcd)
{
      TheJob* walk;
      float Jhr=0;       //work hour of a job
      walk=JobHead;

      FindUrgentJobs(walk, &Jhr, hr, Zcd);
      FindNormalJobs(walk, &Jhr, hr, Zcd);
}


/*********************************************************************/
/*Check and make sure urgent jobs will be scheduled first. Normal    */
/*jobs will not be scheduled unless urgent job queue is empty. Report*/
/*a technician's start location if he has enough time to work on a   */
/*    well scheduled job.                                             */
/*********************************************************************/
int Scheduling(int* Sch, float hr, int mid)
{
      int urgent=TRUE;
      int normal=TRUE;
      int I;
      int x,y;     //x and Y coordinates
      int t;       //work time
      TheJob *walk=NULL;
      walk=UrgHead;
      x=y=t=i=0;

      for(i=0; i<QUART; i++)  //initialize schedule array
            Sch[i]=0;

      if(walk==NULL)            //check if urgent queue is empty
            urgent=FALSE;
      if(walk!=NULL)            //schedule jobs in urgent queue first
            urgent=ScheduleJobs(walk, Sch, hr, mid);
      if(urgent==FALSE){
            walk=NomHead;       //else schedule normal jobs
            normal=ScheduleJobs(walk, Sch, hr, mid);
      }
      if(normal==FALSE){        //if no job is available, exit scheduling
            printf("No job in this area is available by now.\n");
            printf("You can quit here and request jobs in near by
            area.\n");
            return FALSE;
      }
      if(Sch[0]!=0){                          //if jobs are well scheduled,
            FindManLocation(mid, &x, &y); //start job dispatching
            t=FindWkTime(Sch[0]);//find estimated work time of the job
            t=t+TravelTime(Sch[0], &x, &y);//find travel time

            //check if a technician have enough time to work on the job
            if((hr*60)<t){
                printf("No job can be completed in %2.2f hour\n", hr);
                printf("You can go home now. See you tomorrow.\n");
            return FALSE;
            }

            //if he does have enough time, report his start location
            SetStartLocation(Sch[0], mid);
        }
        return TRUE;
}


/*********************************************************************/
/*Schedule jobs by shortest work time first, longest work time first*/
/*and shortest travel distance first respectively. Reschedule these */
/*three queues by genetic algorithm. Find the best schedule from .. */
/*    these six queues and print out the solution.                  */
/*********************************************************************/
int ScheduleJobs(TheJob* Jobhd, int* Sch, float hr, int mid)
{
        FILE* fp;
        int SPT[HALF];      //shortest work time first queue
        int LPT[HALF];      //longest work time first queue
        int SDF[HALF];      //shortest travel distance first queue
        int GSPT[HALF];     //SPT queue rescheduled by genetic algorithm
        int GLPT[HALF];     //LPT queue rescheduled by genetic algorithm
        int GSDF[HALF];     //SDF queue rescheduled by genetic algorithm
        int SUM[HALF];      //array to store work time of above six queues
        int i, s;
        int cnt;            //job counter
        int max;            //maximum work time
        int f=TRUE;         //flag
        int *ptr=NULL;
        TheJob *walk;
        walk=Jobhd;

        fp=fopen("fout", "a");
        s=cnt=i=max=0;      //initialize variables

        while(walk->next!=NULL){//make sure jobs have not been done yet
            if(strcmp(walk->Status, "New")==0)
                SPT[i]=LPT[i]=SDF[i++]=walk->JobID;
            walk=walk->next;
        }

        if(strcmp(walk->Status, "New")==0)
            SPT[i]=LPT[i]=SDF[i++]=walk->JobID;
        cnt=i;              //count number of jobs scheduled

        if(cnt==0)
            return FALSE;
        //build shortest distance first queue and return total work time
        SUM[0]=SDFqueue(SDF, hr, cnt, mid);
        //build shortest work time first queue and return total work time
        SUM[1]=SPTqueue(SPT, hr, cnt, mid, &f);
        //build longest work time first queue and return total work time
        SUM[2]=LPTqueue(LPT, hr, cnt, mid);

        walk=Jobhd;
```

```
i=0;
while(walk->next!=NULL){
      if(strcmp(walk->Status, "New")==0){
            GSPT[i]=SPT[i];   //copy jobs from SPT to GSPT queue
            GLPT[i]=LPT[i];   //copy jobs from LPT to GLPT queue
            GSDF[i]=SDF[i++]; //copy jobs from SDF to GSDF queue
      }
      walk=walk->next;
}

if(strcmp(walk->Status, "New")==0){
      GSPT[i]=SPT[i];
      GLPT[i]=LPT[i];
      GSDF[i]=SDF[i++];
}
//build GSDF queue by genetic algorithm, return total work time
SUM[3]=Genqueue(GSDF, hr, cnt, mid);
//build GSPT queue by genetic algorithm, return total work time
SUM[4]=Genqueue(GSPT, hr, cnt, mid);
//build GLPT queue by genetic algorithm, return total work time
SUM[5]=Genqueue(GLPT, hr, cnt, mid);

max=SUM[0];
for(i=0; i<6; i++){//find the best schedule from these six queues
      if(max<SUM[i]){
            max=SUM[i];
            s=i;
      }
}

if(f==FALSE)
      s=0;

switch(s){
      case 0: ptr=SDF; break;
      case 1: ptr=SPT; break;
      case 2: ptr=LPT; break;
      case 3: ptr=GSDF; break;
      case 4: ptr=GSPT; break;
      case 5: ptr=GLPT; break;
}

//copy the best schedule selected to current working schedule
for(i=0; i< cnt; i++){
      Sch[i]=ptr[i];
      if((i+1)%8==0)
            printf("\n");
}

//print final selected best schedule
fprintf(fp,"POSSIBLE SCHEDULES FOR THE EXPECTED %2.0f MINUTES\n",
hr*60);
fprintf(fp, "%s", "/**********************************/\n");

if(SDF[0]!=0){
fprintf(fp,"%s","Shortest Distance First(SDF) Schedule:\n");
      printf("Shortest Distance First(SDF) Schedule:\n");
```

```
            printQueue(SDF, hr, cnt, mid, fp);
        }
        if(SPT[0]!=0){
            fprintf(fp,"%s","Shortest Processing Time First(SPT)
        Schedule:\n");
            printf("Shortest Processing Time First(SPT) Schedule:\n");
            printQueue(SPT, hr, cnt, mid, fp);
        }
        if(LPT[0]!=0){
            fprintf(fp,"%s","Longest Processing Time first(LPT)
        Schedule:\n");
            printf("Longest Processing Time first(LPT) Schedule:\n");
            printQueue(LPT, hr, cnt, mid, fp);
        }
        if(GSDF[0]!=0){
            fprintf(fp,"%s","SDF Schedule Optimized by Genetic
        Algorithm:\n");
            printf("SDF Schedule Optimized by Genetic Algorithm:\n");
            printQueue(GSDF, hr, cnt, mid,fp);
        }
        if(GSPT[0]!=0){
            fprintf(fp,"%s","SPT Schedule Optimized by Genetic
        Algorithm\n");
            printf("SPT Schedule Optimized by Genetic Algorithm\n");
            printQueue(GSPT, hr, cnt, mid,fp);
        }
        if(GLPT[0]!=0){
            fprintf(fp,"%s","LPT Schedule Optimized by Genetic
        Algorithm\n");
            printf("LPT Schedule Optimized by Genetic Algorithm\n");
            printQueue(GLPT, hr, cnt, mid,fp);
        }
        fclose(fp);
        return TRUE;
}


/**************************************************************************/
/*Print the summary information of the best scheduling solution.    */
/**************************************************************************/
void printQueue(int* T, float hr, int cnt, int mid, FILE* fp)
{
        float Total;          //total work time
        float Wktime;         //work time
        float Trtime;         //travel time
        float t1, t2, t3; //temporary variable for times
        int i;
        int sx, sy;           //coordinates x and y
        int min;              //minutes
        TheMan* step;
        step=ManHead;
        Total=Wktime=Trtime=t1=t2=t3=I=sx=sy=min=0;

        while(step->ManID!=mid)
            step=step->next;

        sx=step->LocX;
        sy=step->LocY;
        min=60*hr;

        //cal. the total work and total travel time of jobs in the queue
        while(t3<min && i<cnt){
                Wktime=Wktime+FindWkTime(T[i]);
                Trtime=Trtime+TravelTime(T[i], &sx, &sy);
                t3=Wktime+Trtime;
                if(t3>min)
                        break;
                printf("#%d, ",T[i]);
                fprintf(fp,"#%d, ",T[i++]);
                t1=Wktime;
                t2=Trtime;
                Total=t1+t2;

        }
        if(t3>min){
                Wktime=t1;
                Trtime=t2;
        }

        //print the summary of the best queue
        printf("\nwork time: %2.0f min.; ", Wktime);
        printf("travel time: %2.0f min.; ", Trtime);
        printf("total %d jobs.\n", i);
        fprintf(fp,"\nwork time: %2.0f min.; ", Wktime);
        fprintf(fp,"travel time: %2.0f min.; ", Trtime);
        fprintf(fp,"total %d jobs.\n", i);
}

/**********************************************************************/
/*Reset a technician's start location when he arrives at a job site.*/
/**********************************************************************/
void SetStartLocation(int jid, int mid)
{
        float miles=0;
        TheMan* Mptr;
        TheJob* Jptr;
        Mptr=ManHead;
        Jptr=JobHead;
        while(Mptr->ManID!=mid)
                Mptr=Mptr->next;
        while(Jptr->JobID!=jid)         //find the job
                Jptr=Jptr->next;
        miles=abs(Mptr->LocX-Jptr->LocX) + abs(Mptr->LocY-Jptr->LocY);
        Jptr->Mileage=miles;            //get mileage
        Mptr->Mileage=Mptr->Mileage+miles;

        Mptr->LocX=Jptr->LocX;  //reset the technician's start location
        Mptr->LocY=Jptr->LocY;
}

/**********************************************************************/
/*Reschedule jobs by genetic algorithm. In a reproduction loop, two */
/*random numbers are generated, jobs in the positions of these two  */
/*numbers are swapped, total work time of jobs in the queue is      */
```

```
/*calculated. The best schedule is picked out and kept from each loop*/
/*********************************************************************/
int Genqueue(int* PT, float hr, int cnt, int mid)
{
      int sx=0, sy=0;    //coordinates x and y
      int wt=0, wt1=0, wt2=0; //work time
      int min=0;         //minutes
      int k=0;
      int len;           //number of jobs
      int total=0;       //total work time
      int r=0,i=0, j=0, tp=0, m=0;
      int temp[HALF];
      TheMan* step;
      step=ManHead;
      for(j=0; j<cnt; j++)
            temp[j]=PT[j];
      while(step->ManID!=mid)          //find the man's work location
            step=step->next;
      sx=step->LocX;
      sy=step->LocY;
      min=(hr*60);

      while(total<min && k < cnt){
            wt2=wt1;
            wt1=wt1+FindWkTime(PT[k]);     //get total work time
            total=total+FindWkTime(PT[k])+TravelTime(PT[k++], &sx,&sy);
      }

      m=REPRODUCTION;
      wt1=wt2;
      if(cnt==1)
            return;
      len=cnt-1;

      while(m--!=0){            //reproduction  loop
            r=(rand() % len); //get random numbers
            i=((rand()) % len);

            tp=temp[r];        //swap jobs in the pos. of these numbers
            temp[r]=temp[i];
            temp[i]=tp;

            total=k=wt2=0;     //initialize variables
            sx=step->LocX;     //get the new start location
            sy=step->LocY;

            while(total<min && k <cnt){    //calculate total work time
                  wt=wt2;
                  wt2=wt2+FindWkTime(temp[k]);
            total=total+FindWkTime(temp[k])+TravelTime(temp[k++],
            &sx,&sy);
            }
            if( wt1<wt){//pick out the best from this reproduction
                  for(j=0; j<cnt; j++)
                        PT[j]=temp[j];
                  wt1=wt;
            }
```

```
        }
        return wt1;
}


/*********************************************************************/
/*Reschedule jobs in the order of their travel distance from        */
/*the technician's start location. Jobs of shortest distance first.*/
/*********************************************************************/
int SDFqueue(int* SDF, float hr, int cnt, int mid)
{
        int i,j,k,n;
        int min;            //minutes
        int wt,wt1;         //work time
        int sx,sy;          //job coordinates x and y
        int d1,d2,total,jid,m;
        int x,y;            //technician coordinates x and y
        int tp[HALF], tp1[HALF];//temporary buffer
        TheMan* step;
        step=ManHead;
        I=j=id=k=sx=sy=min=n=wt1=d1=d2=total=wt=x=y=jid=m=0;

        while(step->ManID!=mid)
                step=step->next;  //find technician's start location
        x=step->LocX;
        y=step->LocY;

        for(i=0; i<cnt; i++)
                tp[i]=tp1[i]=SDF[i];    //put jobs in temp array

        i=j=cnt;
        jid=tp[0];
        while(n<cnt){       //sort jobs in order of travel distance
                d1=TravelDistance(jid, x, y);
                while(j--!=0){
                        if(tp[j]!=0){
                                d2=TravelDistance(tp[j], x, y);
                                if(d2<d1){
                                        d1=d2;
                                        jid=tp[j];
                                }
                        }
                }

                FindJobLocation(jid, &x, &y);
                SDF[n++]=jid;
                for(m=0; m<cnt; m++)
                        if(tp[m]==jid)
                                tp[m]=0;
                jid=0;
                for(m=0; m<cnt; m++)
                        if(tp[m]!=0)
                                jid=tp[m];
                j=cnt;
        }
        x=sx=step->LocX;
        y=sy=step->LocY;
        min=(hr*60);
        m=n=0;
        while(total<min && k<cnt){          //calculate total work time
                wt=wt1;                     //and travel time of jobs
                n=FindWkTime(SDF[k]);       //in this queue
                m=TravelTime(SDF[k++], &sx,&sy);
                wt1=wt1+n;
                total=total+n+m;
        }
        return wt;
}


/**********************************************************************/
/*Calculate the travel distance from the technician's start location*/
/*      to a job.                                                     */
/**********************************************************************/
int TravelDistance(int id, int x, int y)
{
        int EX,EY;   //job coordinates x and y;
        int d;       //distance
        int dx,dy;   //relative distance in direction x and y
        EX=EY=d=dx=dy=0;

        FindJobLocation(id, &EX, &EY);      //find job location

        dx=abs(EX-x);                       //calculate distance
        dy=abs(EY-y);
        d=dx+dy;

        return d;
}

/**********************************************************************/
/*      Search job queue to find job location by a given job ID       */
/**********************************************************************/
void FindJobLocation(int id, int*EX, int*EY)
{
        TheJob *walk;
        walk=JobHead;
        if(id==0)
                return;

        while(walk->JobID!=id)
                walk=walk->next;

        *EX=walk->LocX;
        *EY=walk->LocY;
}

/**********************************************************************/
/*Reschedule jobs in order of actual work time. Jobs of longest       */
/*      work time first.                                              */
/**********************************************************************/
int LPTqueue(int* LPT, float hr, int cnt, int mid)
{
        int i,j,k;
        int min,t,total,wt,wt1;//work time
        int id;            //job ID
        int sx,sy          //technician coordinates x and y
        TheMan* step;
        step=ManHead;
        I=j=id=k=sx=sy=min=t=total=wt=wt1=0;

        while(step->ManID!=mid)
              step=step->next;  //find technician's location

        sx=step->LocX;
        sy=step->LocY;

        for(i=1; i<cnt; i++){    //reschedule jobs in order of work time
              t=FindWkTime(LPT[i]);
              id=LPT[i];

              for(j=i; j>0 && FindWkTime(LPT[j-1])<t; j--)
                    LPT[j]=LPT[j-1];

              LPT[j]=id;
        }

        min=hr*60;
        while(total<min && k<cnt){     //calculate work and travel time
              wt=wt1;                  //of the rescheduled queue
              wt1=wt1+FindWkTime(LPT[k]);
              total=total+FindWkTime(LPT[k])+TravelTime(LPT[k++],
        &sx,&sy);
        }

        return wt;
}


/*******************************************************************/
/*Reschedule jobs in order of their actual work time. Jobs of      */
/*shortest work time first.                                        */
/*******************************************************************/
int SPTqueue(int* SPT, float hr, int cnt, int mid, int* f)
{
        int i,j,k;
        int id;      //job ID
        int sx,sy;   //job coordinates x and y
        int t,total,wt,wt1,min;//work time
        TheMan* step;
        step=ManHead;
        I=j=id=k=sx=sy=min=t=total=wt=wt1=0;

        while(step->ManID!=mid)
              step=step->next;  //find technician's location
        sx=step->LocX;
        sy=step->LocY;

        for(i=1; i<cnt; i++){    //reschedule jobs
              t=FindWkTime(SPT[i]);
              id=SPT[i];
```

```
            for(j=i; j>0 && FindWkTime(SPT[j-1])>t; j--)
                SPT[j]=SPT[j-1];
            SPT[j]=id;
        }

        min=(hr*60);
        while(total<min && k<cnt){      //calculate total work time
            wt=wt1;                     //and travel time of the
                                    //rescheduled queue
            wt1=wt1+FindWkTime(SPT[k]);
            total=total+FindWkTime(SPT[k])+TravelTime(SPT[k++],
            &sx,&sy);
        }

        if(total < min)
            *f=FALSE;
        return wt;
}


/*********************************************************************/
/*Calculate travel time from a technician to a job.                  */
/*********************************************************************/
int TravelTime(int id, int* sx, int* sy)
{
        int EX,EY;  //job coordinates x and y
        int t;      //work time
        int dx,dy;  //relative distance in direction of x and y
        EX=EY=t=dx=dy=0;

        FindJobLocation(id, &EX, &EY);
        dx=abs(EX-*sx);
        dy=abs(EY-*sy);
        t=60*(dx+dy)/SPEED;
        *sx=EX;
        *sy=EY;

        return t;
}


/*********************************************************************/
/*    Return the work time of a job.                                 */
/*********************************************************************/
int FindWkTime(int JID)
{
        TheJob *walk;
        float t=0;  //work time
        int min=0;  //minutes
        walk=JobHead;

        if(walk==NULL)
            return 0;

        while(walk->JobID!=JID)
            walk=walk->next;
        t=walk->EWkTime;
        min=(t*60);

        return min;
}

/*********************************************************************/
/*Pick urgent jobs out from job list and build an urgent job queue. */
/*********************************************************************/
void FindUrgentJobs(TheJob* JobH, float*Jhr, float hr, int Zcd)
{
        int Urg, New;
        TheJob* walk;
        walk=JobH;

        while(walk->next!=NULL && *Jhr <= hr*10){
                Urg=strcmp(walk->Type,"Urgent");
                New=strcmp(walk->Status,"New");

                //pick out the new urgent jobs
                if(Zcd==walk->ZipCode && Urg==0 && New==0){
                        *Jhr=*Jhr+walk->EWkTime;
                        NewJobList(walk); //build urgent job list
                }
                walk=walk->next;
        }

        Urg=strcmp(walk->Type,"Urgent");
        New=strcmp(walk->Status,"New");
        if(Zcd==walk->ZipCode && Urg==0 && New==0 && *Jhr <= hr*2){
                *Jhr=*Jhr+walk->EWkTime;
                NewJobList(walk);
        }
}

/*********************************************************************/
/*Pick normal jobs out from job list and build an normal job queue. */
/*********************************************************************/
void FindNormalJobs(TheJob* JobH, float*Jhr, float hr, int Zcd)
{
        int Urg, New;
        TheJob* walk;
        walk=JobH;

        while(walk->next!=NULL && *Jhr<=hr*10){
                Urg=strcmp(walk->Type,"Urgent");     //find out normal jobs
                New=strcmp(walk->Status,"New");
                if(Zcd==walk->ZipCode && Urg!=0 && New==0){
                        *Jhr=*Jhr+walk->EWkTime;
                        NewJobList(walk);         //build normal job queue
                }
                walk=walk->next;
        }

        Urg=strcmp(walk->Type,"Urgent");
        New=strcmp(walk->Status,"New");
        if(Zcd==walk->ZipCode && Urg!=0 && New==0 && *Jhr <= hr*2){
                *Jhr=*Jhr+walk->EWkTime;
                NewJobList(walk);
        }
}

/*********************************************************************/
/*Reset job parameters to build a urgent or normal job queue.        */
/*********************************************************************/
void NewJobList(TheJob* wk)
{
        TheJob* Ptr;
        Ptr = ((TheJob*)malloc(sizeof (TheJob)));
        Ptr->JobID=wk->JobID;
        Ptr->ZipCode=wk->ZipCode;
        strcpy(Ptr->Type, wk->Type);
        strcpy(Ptr->Status, wk->Status);
        Ptr->EWkTime=wk->EWkTime;
        Ptr->AWkTime=wk->AWkTime;
        strcpy(Ptr->DueDay, wk->DueDay);
        strcpy(Ptr->StTime, wk->StTime);
        strcpy(Ptr->FinTime, wk->FinTime);
        strcpy(Ptr->StDate, wk->StDate);
        strcpy(Ptr->FinDate, wk->FinDate);
        strcpy(Ptr->Address, wk->Address);
        Ptr->LocX=wk->LocX;
        Ptr->LocY=wk->LocY;
        Ptr->Mileage=wk->Mileage;
        Ptr->next=NULL;
        LinkNewMe(Ptr);
}


/*********************************************************************/
/*      Building urgent or normal job queue.                         */
/*********************************************************************/
void LinkNewMe(TheJob* Pt)
{
        TheJob* walk=NULL;

        if(strcmp(Pt->Type, "Urgent")==0){
                if(UrgHead==NULL){
                        UrgHead=Pt;
                        return;
                }
                walk=UrgHead;
        }
        else{
                if(NomHead==NULL){
                        NomHead=Pt;
                        return;
                }
                walk=NomHead;
        }

        while(walk->next != NULL)
                walk=walk->next;
        walk->next=Pt;
```

```
}

/**********************************************************************/
/*Check login name and employee ID. If both are correct, this person */
/*is accepted and authorized to work on the position. The person's   */
/*login time is recorded by the system as start time.                */
/**********************************************************************/
int LoginAccept(int id)
{
	FILE* fp;
	time_t t1;		//time
	char *s;		//time pointer
	int flag=FALSE;
	TheMan* mptr;

	fp=fopen("fout","a");

	if((t1=time((time_t*)0))!=(time_t)-1)//get system time
		s=ctime(&t1);
	else
		printf("Error time. \n");

	mptr=ManHead;
	flag=IsNameinList(id);
	if(flag==FALSE)
		return FALSE;

	while(mptr->ManID != id)	//find the technician
		mptr=mptr->next;

	strcpy(mptr->LoginTime, s);	//print out login information
	printf("Name:           %s\n", mptr->Name);
	printf("Employee ID:  %d\n", mptr->ManID);
	printf("Work zipcode: %d\n", mptr->ZipCode);
	printf("login at:      %s", mptr->LoginTime);
	fprintf(fp, "%s", "/**********************************/\n");
	fprintf(fp,"Name:           %s\n", mptr->Name);
	fprintf(fp,"Employee ID:  %d\n", mptr->ManID);
	fprintf(fp,"Work zipcode: %d\n", mptr->ZipCode);
	fprintf(fp,"login at:      %s", mptr->LoginTime);

	fclose(fp);
	return TRUE;
}


/**********************************************************************/
/*check if a person is in company's workforce list.                 */
/**********************************************************************/
int IsNameinList(int id)
{
	int flag=FALSE;
	TheMan* mptr;
	mptr=ManHead;

	if(mptr==NULL)
		return FALSE;
```

```
        while(mptr->next!=NULL){//check the person by his employee ID
                if(mptr->ManID == id)
                        flag=TRUE;
                mptr=mptr->next;
        }
        if(mptr->ManID == id)
                flag=TRUE;
        return flag;
}


/*******************************************************************/
/*when a technician logout, this function will record his         */
/*logout time and work summary. This information is  printed out.  */
/*******************************************************************/
void LogoutAccept(int id)
{
        FILE* fp;
        time_t t1;              //time
        char *str;              //time pointer
        int flag=FALSE;
        TheMan* mptr;
        fp=fopen("fout","a");

        if((t1=time((time_t*)0))!=(time_t)-1)//get logout time
                str=ctime(&t1);
        else
                printf("Error time. \n");
        mptr=ManHead;
        while(mptr->ManID!=id)
                mptr=mptr->next;  //find the technician

        strcpy(mptr->LogoutTime, str);          //record logout time
        printf("%s ", mptr->Name);              //print logout information
        printf("logout at: %s \n", str);
        fprintf(fp,"%s ", mptr->Name);
        fprintf(fp,"logout at: %s \n", str);

        fclose(fp);
}


/*******************************************************************   /
/*When a technician arrives on site, this function will record the   */
/*time and date to calculate his drive time. This information is also*/
/*      printed out.                                                 */
/*********************************************************************/
void ArriveOnSite(int id, char* St, char* Sd)
{
        FILE* fp;
        time_t t1;              //time
        struct tm *tptr;        //time pointer
        TheJob* Jptr;

        fp=fopen("fout","a");
```

```
	if((t1=time((time_t*)0))!=(time_t)-1)
		tptr=localtime(&t1);	//get system time
	else
		printf("Error time. \n");

	Jptr=JobHead;
	if(Jptr==NULL)
		return;

	while(Jptr->JobID != id)
		Jptr=Jptr->next;		//get the job

	if(tptr->tm_year < 100)
		tptr->tm_year=1900 + tptr->tm_year;
	else
		tptr->tm_year=2000 + tptr->tm_year;

	//record system time and date
sprintf(Jptr->StTime, "%d:%d:%d", tptr->tm_hour,tptr->tm_min,tptr-
>tm_sec);
sprintf(Jptr->StDate, "%d/%d/%d", tptr->tm_mon+1,tptr->tm_mday,tptr-
>tm_year);
	strcpy(St, Jptr->StTime);
	strcpy(Sd, Jptr->StDate);
	printf("job #%d start date: %s \n", id, Jptr->StDate);
	printf("job #%d start time: %s \n", id, Jptr->StTime);
	fprintf(fp,"job #%d start date: %s \n", id, Jptr->StDate);
	fprintf(fp,"job #%d start time: %s \n", id, Jptr->StTime);

	fclose(fp);
}

/*********************************************************************/
/*when a job is done, this function record the date and time the    */
/*job is done, change status to DONE and. This information is also */
/*    printed out.                                                   */
/*********************************************************************/
void JobDone(int id, char* Ft, char* Fd)
{
	FILE* fp;
	time_t t1;		//time
	struct tm *tptr;	//time pointer
	char *s;
	TheJob* Jptr;

	fp=fopen("fout","a");

	if((t1=time((time_t*)0))!=(time_t)-1){
		s=ctime(&t1);
		tptr=localtime(&t1);		//get system time
	}
	else
		printf("Error time. \n");

	Jptr=JobHead;
	if(Jptr==NULL)
		return;
	while(Jptr->JobID != id)	//get the job
		Jptr=Jptr->next;

	if(tptr->tm_year < 100)
		tptr->tm_year=1900 + tptr->tm_year;
	else
		tptr->tm_year=2000 + tptr->tm_year;

	//record time and date
sprintf(Jptr->FinTime, "%d:%d:%d", tptr->tm_hour,tptr->tm_min,tptr-
>tm_sec);
sprintf(Jptr->FinDate, "%d/%d/%d", tptr->tm_mon+1,tptr->tm_mday,tptr-
>tm_year);
	strcpy(Ft, Jptr->FinTime);
	strcpy(Fd, Jptr->FinDate);
	printf("Job #%d complete date: %s \n", id, Jptr->FinDate);
	printf("Job #%d complete time: %s \n", id, Jptr->FinTime);
	fprintf(fp,"Job #%d complete date: %s \n", id, Jptr->FinDate);
	fprintf(fp,"Job #%d complete time: %s \n", id, Jptr->FinTime);
	fclose(fp);
	ChangeStatus(JobHead, id);
	ChangeStatus(UrgHead, id);
	ChangeStatus(NomHead, id);
}
/**********************************************************************/
/*when a job is done, this function will change job status to DONE. */
/**********************************************************************/
void ChangeStatus(TheJob* HD, int id)
{
	TheJob *walk;
	walk=HD;

	if(HD==NULL)
		return;

	while(walk->next!=NULL){	//find the job
		if(walk->JobID==id)
			strcpy(walk->Status,"Done");	//change status to Done
		walk=walk->next;
	}

	if(walk->JobID==id)
		strcpy(walk->Status,"Done");
}

/**********************************************************************/
/*Input start time start date and finish time finish date. Total     */
/*    duration of time is calculated and returned.                   */
/**********************************************************************/
int TimeDuration(char* St, char* Sd, char* Ft, char* Fd)
{
	long Minutes;		//minutes
	long AbEnd;		//absolute Julian end date number
	long AbStart;		//absolute Julian start date number
	short EndHr;		//end hour
	short EndMin;		//end minute
```

```
        int EndDayMM;              //total minutes in end day
        long LapDay;               //total duration of days
        short StartHr;             //start hour
        short StartMin;            //start minutes
        short StartDayMM;          //total minutes in start day
        char StTime[QUART];        //start time string
        char FinTime[QUART];       //finish time string
        char* str;

        strcpy(StTime, St);
        strcpy(FinTime, Ft);

        AbStart=AbsJulian(Sd);  //get Julian day number
        AbEnd=AbsJulian(Fd);

        str=strtok(StTime, ":\0"); //pick hr and min.
        StartHr=atoi(str);
        str=strtok(NULL, ":\0");
        StartMin=atoi(str);

        str=strtok(FinTime, ":\0"); //pick hour and minute.
        EndHr=atoi(str);
        str=strtok(NULL, ":\0");
        EndMin=atoi(str);

        StartDayMM=(StartHr*60)+StartMin;     //get total min. in start day
        EndDayMM=(EndHr*60)+EndMin;           //get total min. in end day

        if(AbStart=AbEnd)
                Minutes=EndDayMM-StartDayMM; //calculate time duration
        else{
                if(AbEnd>AbStart){
                        LapDay=AbEnd-AbStart;
                        Minutes=LapDay*1440+EndDayMM-StartDayMM;
                }
                else
                        Minutes=0;
        }

        return Minutes;
}

/*************************************************************************/
/*Get technician's location and return the value back in *EX and *EY.*/
/*************************************************************************/
void FindManLocation(int id, int*EX, int*EY)
{
        TheMan *walk;
        walk=ManHead;

        while(walk->ManID!=id)
                walk=walk->next;

        *EX=walk->LocX;
        *EY=walk->LocY;
}
```

```
/**********************************************************************/
/*Check if the input file is empty or if the file contains invalid  */
/*characters such as tabs, blanks and new line characters, then call */
/*InputJob() and InputMan() to read in the input data.              */
/**********************************************************************/
int manageInput()
{
      FILE *fp,*wfp;
      char T[FULL], EXIT[QUART];
      int i, flag=0;

      if(( fp = fopen("Job.dat","r")) == NULL){
            printf("Can't open file!\n");
            exit(0);
      }

      for(i=0;i<FULL;i++)
            T[i]='\0';          //clean the buffer

      while((fgets(T, FULL, fp))!=NULL){  //check the validity of input
            for(i=0; i<FULL;i++)
                  if(T[i]=='\t'||T[i]==' '||T[i]=='\n'||T[i]=='\0')
                        ;
                  else
                        flag =1;
      }

      if(flag==0){          //print alert message and exit
            printf("This file is empty. Type EXIT to quit!\n");
            scanf("%s", EXIT);
            while(strcmp(EXIT, "EXIT")!=0){
                  printf("\nType EXIT to quit!\n");
                  scanf("%s", EXIT);
            }
            return FALSE;
      }

      fclose(fp);
      wfp=fopen("fout", "w"); //print out put title
      fprintf(wfp, "%s\n", "     ALL INPUT JOB LIST");
      fprintf(wfp, "%s","/**********************************/\n");

      fclose(wfp);

      InputJob();          //call to read input data
      InputMan();

      wfp=fopen("fout", "a");

      fprintf(wfp, "%s\n", "     JOB LIST SORTED BY DUE DAY");
      fprintf(wfp, "%s","/**********************************/\n");

      fclose(wfp);

      PrintJobs(JobHead);       //call to print all input jobs
      PrintMan();               //call to print all input technician data
      return TRUE;
```

```
}


/*********************************************************************/
/*Check the validity of technician's assigned area zip code.        */
/*********************************************************************/
int IsZipCodeValid(int zp, int id)
{
      TheMan* walk;
      walk=ManHead;

      while(walk->ManID!=id)
            walk=walk->next;  //find the technician

      if(walk->ZipCode==zp)   //check if input zip code is valid
            return TRUE;
      else if (zp==100)
            return 100;
      else if(zp== 200)
            return 200;
      else if(zp==300)
            return 300;
      else if(zp==400)
            return 400;
      else
            return FALSE;
}


/*********************************************************************/
/*Calculate the sum of work time and travel time.                   */
/*********************************************************************/
float getWktime(int jid)
{
      float tm=0;
      TheJob* walk;
      walk=JobHead;

      while(walk->JobID!=jid)
            walk=walk->next;

      tm=tm+walk->AWkTime;
      tm=tm+walk->Mileage/SPEED;

      return tm;
}


/*********************************************************************/
/*main()simulates the operation screen. It calls to read and mnage  */
/*input, prompts login and logout information, call to schedule jobs*/
/*check the validity of input, provide an interface between the user*/
/*    and the system.                                               */
/*********************************************************************/
main()
{
  int i,x,y;
int mx,my;              //coordinates x and y
int Eid,Jid;            //employee ID, Job ID
int Accept=FALSE;       //flag
int zp;                 //zip code
char wk[QUART];         //select job flag
char EXIT[QUART];       //exit function flag
char Site[QUART];       //arrive on site flag
char Done[QUART];       //job done flag
float hr=0;             //requested work time
float tm=0;             //calculated work time
float TT=0;             //total work time
float WT=0;             //one job work time
char St[QUART];         //start time
char Sd[QUART];         //start date
char Ft[QUART];         //finish time
char Fd[QUART];         //finish date
Chg[QUART];             //temp buffers
int Sch[QUART];         //change work time flag

i=x=zp=y=mx=my=Eid=Jid=0;//initialize variables

if(manageInput()==FALSE)      //read input
    exit(0);

printf("Type your employee ID to login: ");
    scanf("%d", &Eid);

Accept=LoginAccept(Eid);

//if login is incorrect, retype ID number and check again
while(Accept!=TRUE){
  printf("\nError! Retype your employee ID to login: ");
  scanf("%d", &Eid);
  Accept=LoginAccept(Eid);
}

FindManLocation(Eid, &mx, &my);    //get technician' location
x=mx;     y=my;

//check his request
printf("\nRequest a job? Type YES! Quit? Type NO! ");
scanf("%s", wk);
while(strcmp(wk, "YES")!=0 && strcmp(wk, "NO")!=0){
  printf("\nRequest a job? Type YES! Quit? Type NO! ");
  scanf("%s", wk);
}

if(strcmp(wk, "NO")==0)
  exit(0);

//check technician's input zip code
printf("\nType your work zipcode: ");
scanf("%d", &zp);
while(IsZipCodeValid(zp, Eid)==FALSE){
  printf("\nError! Retype your work zipcode: ");
  scanf("%d", &zp);
}
```

```
  if(IsZipCodeValid(zp, Eid)!=TRUE)
  printf("Area %d is beyond your home area, you will have a long
drive!\n", zp);

  printf("\nHow many hours do you expect to work today: ");
  scanf("%f", &hr);      //get his requested work time

  ScheduleList(hr, zp); //schedule jobs for the technician

  while(strcmp(wk,"YES")==0){
    strcpy(wk,"  "); strcpy(Site,"  "); strcpy(Done,"  ");
      while(hr==0){
   printf("Work hours can not be zero. Retype work time or type 9999 to
      quit!");
      scanf("%f", &hr);
      if(hr==9999)
          break;
      }

      //while loop continue to dispatch jobs for the technician
      //if he request to work over time, the system allow him to do so.
      while(Scheduling(Sch, hr, Eid)==FALSE){
        printf("Type EXIT to quit! or WORK to request a over time
            job!\n");
        scanf("%s", EXIT);
        while(strcmp(EXIT, "EXIT")!=0 && strcmp(EXIT, "WORK")!=0){
          printf("Type EXIT to quit! or WORK to request a over time
            job!\n");
          scanf("%s", EXIT);
        }

        if(strcmp(EXIT, "EXIT")==0)
          break;

        if(strcmp(EXIT, "WORK")==0){
          printf("\nHow many hours do you expect to work next: ");
          scanf(" %f", &hr);
        }
      }

      if(strcmp(EXIT, "EXIT")==0)
        break;

      printTheJob(Sch[0], &x, &y);  //print the scheduled job for him
      printf("\nWhen arrives at site, type SITE  ");
      scanf("%s", Site);
      while(strcmp(Site, "SITE")!=0){
        printf("\nWhen arrives at site, type SITE  ");
        scanf("%s", Site);
      }

      ArriveOnSite(Sch[0], St, Sd);//check man on site

      printf("\nWhen job is completed, type DONE  ");
      scanf("%s", Done);
      while(strcmp(Done, "DONE")!=0){
        printf("\nWhen job is completed, type DONE  ");
        scanf("%s", Done);
      }

      JobDone(Sch[0], Ft, Fd);       //if job is done, mark job done

      tm=TimeDuration(St,Sd,Ft,Fd);//calculate work time

      SetWkTime(Sch[0],Eid, tm);     //record work time

      WT=getWktime(Sch[0]);

      TT=TT+WT; hr=hr-WT;      //calculate remaining time in the day
      printf("Remaining time %2.2f hours.\n", hr);
      printf("This job takes %2.2f hours.\n", WT);
      printf("Total time used %2.2f hours.\n", TT);

      //continue the loop, if he request another job, go on
      printf("\nRequest a job? Type YES! Quit? Type NO! ");
      scanf("%s", wk);
      while(strcmp(wk, "YES")!=0 && strcmp(wk, "NO")!=0){
        printf("\nRequest a job? Type YES! Quit? Type NO! ");
          scanf("%s", wk);
      }

      //check if he request more work time or less work time
      //make a change if he likes
      if(strcmp(wk,"YES")==0){
        printf("\nChange work time? Type YES! NO change? Type NO! ");
        scanf("%s", Chg);
        while(strcmp(Chg, "YES")!=0 && strcmp(Chg, "NO")!=0){
          printf("\nChange work time? Type YES! NO change? Type NO! ");
          scanf("%s", Chg);
        }
        if(strcmp(Chg, "YES")==0){
          printf("\nHow many hours do you expect to work next: ");
          scanf(" %f", &hr);
        }
      }
      tm=0;
  }

  LogoutAccept(Eid);      //if he finishes his job for the day, log out
  printTheMan(Eid);       //print the day's work report
}
```

VITA 2

Shengli Cao

Candidate for the Degree of

Master of Science

Thesis: A SURVEY OF ALGORITHMS FOR SCHEDULING
NON-INTERRUPTIBLE TASKS

Major Field: Computer Science

Biographical:

Personal Data: Born in Yongji, Shanxi, P.R. China, August 21, 1958, the son of Shilong Cao and Chunxian Xu.

Education: Graduated from Yongji High School, Yongji, Shanxi, P.R. China in 1976; received Bachelor of Engineering degree in Civil Engineering from Taiyuan Institute of Technology, P.R. China in July 1982; completed the requirements for the Master of Science degree at Oklahoma State University in July 2000.

Experience: Graduate teaching assistant of Computer Science Department of Oklahoma State University, August 1999 to December 1999. Employed as a computer lab assistant, August 1999 to November 1999. Employed as a senior engineer by BERIS from 1982 to 1997