To the Graduate Council:

I am submitting herewith a dissertation written by William John Plotnicki entitled "Periodic Dynamic Scheduling with Application to Academic Course Scheduling." I recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Management Science.

Robert S. Garfinkel, Major Professor

We have read this dissertation
and recommend its acceptance:

Donald J. Wheeler

Richard E. Rosenthal

Colin E Bell

Accepted for the Council:

Vice Chancellor
Graduate Studies and Research

B

PERIODIC DYNAMIC SCHEDULING WITH APPLICATION TO ACADEMIC COURSE SCHEDULING

A Dissertation

Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

William John Plotnicki

August 1980

ABSTRACT

In an academic institution, the problem of scheduling courses to quarters is a difficult task. In fact, because of its complexity, many administrators are satisfied with just a feasible schedule in which all courses offered can be staffed. This work addresses an additional dimension of the problem by determining a schedule such that the time students require to graduate is minimized, while still maintaining a feasible schedule with regards to staffing.

To accomplish this task, an interactive computer simulation model was developed which captured most of the complexities of the general academic course schedule problem. The program was executed on data from The University of Tennessee/Knoxville MBA Program, and results and recommendations for course schedules are presented.

The general academic course scheduling problem also motivated a new class of scheduling problems. For a meaningful course schedule, it is usually the case that the same set of courses will be offered every fall, another set of courses every winter, and so on.

To be more precise, if S_k is the set of courses offered in period k, $k \in \{0,1,2,\ldots\}$, then there exists a positive integer n such that

$$S_k = S_{[k]_n}$$

where $[k]_n$ is k modulo n. Such a schedule is called a periodic schedule.

There are other situations in which periodic schedules are also applicable. In an industrial situation where jobs arrive on a periodic

basis, restrictions to periodic schedules may be valid. The last part of this work deals with the relation of periodic scheduling to classical scheduling and with the development of optimal algorithms for some special cases of periodic scheduling problems.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE PAGE

FIGURE PAGE

CHAPTER 1

INTRODUCTION

There are many scheduling problems that arise in course scheduling in an academic institution. Students would like to construct course schedules which would enable them to graduate as quickly as possible. Administrators need to schedule courses at times when instructors and classrooms are available. They must also decide in which quarters to offer a course if it can only be offered once or twice a year due to staffing constraints.

The problem of course scheduling from the administrative side is certainly a monumental task. Many administrators, after weeks of work, would be happy to settle for a feasible course schedule in which all courses offered can be staffed. However, there is another dimension to the scheduling problem.

Most administrators have been confronted by students needing to take a particular course in a quarter in which either the course is not offered or it is full. As a result, the students are forced in some cases to either delay graduation or take less desirable substitute courses.

A well-designed ongoing program will, in some sense, minimize the number of such scheduling problems that occur; but the process of such a program evolving is time consuming, and many students may encounter delays. The problem facing the administrator then is to schedule courses to quarters in such a way that all staffing constraints are

satisfied and some measure of the time students require to graduate is minimized.

Before attempting a solution for the above scheduling problem, it is useful to list some of the aspects of the problem which make it complex.

1. The number of courses to be scheduled is generally large.
2. The number of possible curricula that students can follow is large.
3. The prerequisite and co-requisite structure of courses are completely general.
4. The number of courses students take per quarter varies from student to student and quarter to quarter.
5. Many students who start a program drop out at various times throughout their program.
6. Transfer students enter a program with credit for some of the courses in their curricula.

The list is by no means a complete list of the complications involved in course scheduling, but is sufficient to convey the difficulty of the problem. The literature (see e.g. [3], [5], [10]) reveals that other authors have generally settled for inexact algorithms even for problems having fewer complications than those listed above; consequently, an optimal solution would be extremely difficult to obtain.

However, the problem facing the administrator is real and still exists. In fact, this entire work was motivated by such a problem. In September of 1979, the MBA program at The University of Tennessee was

completely redesigned, and administrators were faced with the problem of scheduling courses to quarters. To aid the administration with their scheduling problem, an interactive simulation model was developed which captured most of the complexities of the general problem. Chapters 3, 4, and 5 deal with the design and implementation of this model.

Even though the general academic course scheduling problem is probably intractable, it did motivate a new class of scheduling problems. For a meaningful course schedule, it is usually true that the same set of classes will be offered every fall, another set of classes offered every winter and so on.

In some programs, courses are offered less frequently than once a year. For example, the same set of courses is offered every other fall. Nevertheless, the following basic property is the same. There exists a cycle during which course offerings repeat themselves. To be precise, if S_k is the set of courses offered in period k, $k \in \{(0,1,2,\ldots\}$, then there exists a positive integer n such that

$$S_k = S_{[k]_n}$$

where $[k]_n$ is k modulo n. Such a schedule is called a periodic schedule. Academic course schedules are generally periodic with only minor deviations.

There are other situations in which periodic schedules are also applicable. In an industrial situation where jobs arrive on a periodic basis, restrictions to periodic schedules may be valid.

If the period length n is large, $n \to \infty$, the periodic property becomes irrelevant; and the class of periodic scheduling problems is the same as the classical scheduling problems. Consequently, this work will consider problems for which n is fixed and relatively small.

Chapters 6-9 will be concerned with the relation of periodic scheduling to classical scheduling and with the development of optimal algorithms for some special case of periodic scheduling problems.

CHAPTER 2

REVIEW OF THE LITERATURE

A. INTRODUCTION

Scheduling can be viewed as the allocation of resources to perform a collection of tasks, where it is generally required that some performance measure is optimized. This rather broad definition gives an indication of the large variety of problems that could be considered scheduling problems. A general overview of scheduling theory can be found in several recent books [3] [5] [22].

In most scheduling problems, the resources can be considered to be machines or processors and the collection of related tasks are called jobs. Because of the large variety of scheduling problems, they are usually divided into many categories. If the set of tasks to be processed does not change over time, the problem is static. If new tasks arrive over time, then the system is said to be dynamic. In addition, if the arrival times of new jobs are not known with certainty, the model is stochastic. In this work, only dynamic deterministic models will be considered where all arrival times are known with certainty.

Sometimes the tasks that comprise a job must be performed in a certain order or at least a partial order. This ordering is called the precedence structure, and problems can be categorized according to the nature of the precedence structure.

Another distinction made between scheduling problems is the number of processors available. This parameter tends to cause considerable differences in the difficulty of solution. For example, it may be easy to solve a particular scheduling problem where only two processors are available and the same problem with three processors may be intractable.

There are many other possible parameters of scheduling problems and the various combinations of parameters contribute to the richness of the scheduling theory literature.

B. PROBLEM COMPLEXITY

The issue of complexity is certainly not new, but only recently has it been rigorously defined. Everyone has compared two problems and concluded that one problem is harder to solve than another. But harder in what sense? The pioneering work by Cook [8] divided problems into classes dependent on their complexity. A complete treatment of the area of problem complexity can be found in Garey and Johnson [10].

Before discussing the various classes of problems, some background needs to be developed on how to measure the complexity of a problem. To begin, a problem is just a question to be answered. A problem can be completely described by giving a general description of the parameters and what properties the solution is required to satisfy. When the parameters are given specific values, it is called an instance of the problem. There are many ways an instance of a problem may be described. These methods are called encoding schemes. For complexity results, the encoding scheme should be concise or contain no unnecessary information.

Also, all numbers should be represented in binary or any number system other than base one. Given that an encoding scheme has been picked that satisfies the above criteria, the number of symbols required to represent an instance of a problem is called the input length. Let n be the input length of a problem. An algorithm is a polynomial time algorithm if there exists a polynomial function p(n) such that a solution can always be obtained in time less than or equal to p(n). If no such polynomial function exists, then the algorithm is said to have exponential time complexity. Notice that time complexity is a worst-case measure. For example, if a problem has complexity f(n), then at least one problem instance requires f(n) time. Many other problem instances may require far less time than f(n) to solve.

The theory of problem complexity divides problems into classes depending on their problem complexity. There are two classes of problems of particular interest: P and NP. Membership in P requires that a problem can be solved in polynomial time on a deterministic computer, while membership in NP means that a problem can be solved in polynomial time on a nondeterministic computer. One way to view a nondeterministic computer is that it has the power to create a copy of itself. As a result, when a nondeterministic computer is trying to solve a problem using an enumeration tree, at every branch it has the power to duplicate itself. Therefore, the width of the tree is unimportant. The depth of the tree must be bounded by a polynomial of the input length.

A problem p is NP-complete if $p \in NP$ and every problem $p' \in NP$ reduces to p. A problem p_1 is said to reduce (polynomially) to p_2 ($p_1 \propto p_2$) if

for any instance of p_1 an instance of p_2 can be constructed in polynomial time such that solving the instance of p_2 will solve the instance of p_1. The implication of the class of problems that are NP-complete is that if there exists a polynomial algorithm for any problem in this class, there exists a polynomial algorithm for all problems in NP. Thus would imply P = NP. Because of the length of time researchers have spent trying to solve problems that are NP-complete, it is generally considered unlikely that P = NP.

There is another encoding scheme which is of theoretical interest called unary encoding. In unary encoding, the number n would be represented by n ones. The reason for the distinction between binary and unary encoding is that there are several problems that are NP-complete when binary encoding is used and yet polynomial bounded under unary encoding [11] [17]. The knapsack problem is a classic example of this situation. Algorithms that are polynomially bounded under unary encoding are sometimes called pseudopolynomial algorithms.

When looking at the various encoding schemes, the strongest results then are to show a problem to be polynomially bounded if binary encoding is used or NP-complete under unary encoding. Recently in scheduling literature, more NP-completeness arguments are using the stronger unary encoding results ([9], [11]).

C. GENERAL SCHEDULING PROBLEM

There are many types of scheduling problems. As a result, it is difficult to define the general problem of which all other scheduling

problems are a special case. However, the majority of scheduling problems do fit into the following scheme.

Job Structure

Consider a set of jobs $J = \{J_1, J_2, \ldots, J_n\}$ that need to be processed. Further, each job can be divided into subjobs or tasks. Let T denote the set of all tasks $\{T_1, T_2, \ldots, T_q\}$. A job then is a subset of T. For example, $J_2 = \{T_2, T_4, T_6\}$ defines job two to include tasks two, four, and six. Also, there may exist a partial ordering of tasks $<$ such that if $T_i < T_j$ then T_j cannot begin execution until T_i is completed. Such a partial ordering, called the precedence structure, can be depicted by a directed acyclic graph. There are three special cases of precedence structures which are often considered: empty, tree, and arbitrary. An empty precedence structure means that there is no required ordering of tasks. In a tree precedence structure, each task has either a unique predecessor or a unique successor task. For example, in Figure 2.1 (a), each task has a unique predecessor. This type of tree is called a branching tree. Figure 2.1 (b) is an example of an assembly tree where each task has a unique successor.

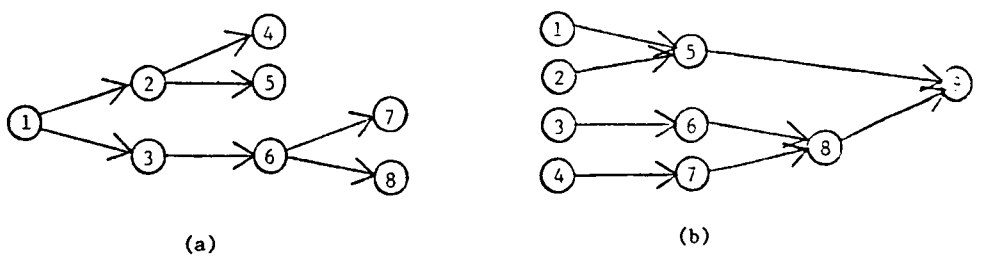

Figure 2.1. Tree Precedence Structures.

The arbitrary precedence structure can be any directed graph as long as there are no directed cycles. An example of an arbitrary structure is given in Figure 2.2.

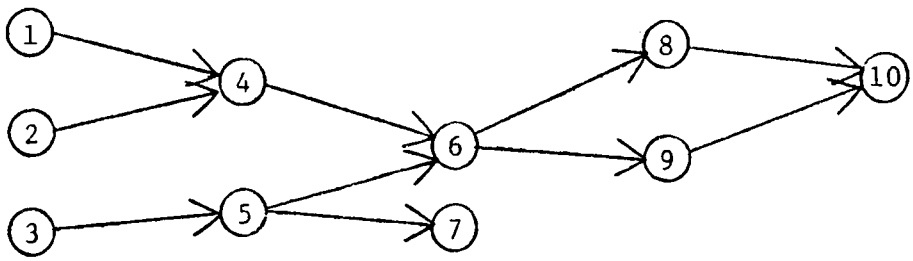

Figure 2.2. Arbitrary Precedence Structure.

It may be the case that not all jobs are available for processing at the same time. Let r_j be the time job j is ready to be processed. If $r_j = 0$ for all j, the problem is static; otherwise, it is said to be dynamic. If all r_j are not known in advance, the problem is stochastic. Only deterministic problems, where all parameters are known in advance, will be considered in this work. Also, each job may have a due date associated with it. Let d_j be the time when job j is due to be completed.

Once a task has started execution, several situations can occur. The task must run to completion or it can be preempted by another task. These two categories are called nonpreemptive and preemptive scheduling respectively.

Processor Structure

Let P be the set of processors $\{P_1, P_2, \ldots, P_m\}$. In general, the processors need not be identical. They may differ as to speed and which

tasks each can execute. Let P(i) be the set of tasks processor i can execute. For example, if $P(3) = \{T_4\}$, then processor three can execute only task four. Also, the processors may be able to execute more than one task simultaneously. Let N_i be the maximum number of tasks that processor i can execute simultaneously. In most scheduling problems, $N_i = 1$. Since the processors need not be identical, let ϵ_{ij} be the time required for processors i to execute task j. In the case of identical processors, only one subscript is required giving the task number.

If a processor is currently running task i, it may take a nonzero amount of time to reconfigure the processor so that it can run task j. This time is called the setup time. Nonzero setup times cause great difficulty in scheduling problems and in this work all setup times will be zero.

Performance Measures

In creating a schedule, one is usually trying to optimize some measure of the schedule's effectiveness called a performance measure. After a schedule has been determined, the following measures are available for each job.

(Completion time) C_j = the completion time for job j

(Flow time) F_j = the time job j spends in the system $(C_j - r_j)$

(Lateness) L_j = the time by which the completion time of job j exceeds its due date $(C_j - d_j)$

(Tardiness) T_j = the lateness of job j if it fails to meet its due date, zero otherwise (max $\{L_j, 0\}$)

From these individual job measures, aggregate performance measures can be calculated. Suppose that there are m jobs. In addition, each job is given a measure of importance w_j. Then the following performance measures can be calculated.

Mean Weighted Flowtime $$\overline{F}_w = \sum_{j=1}^{m} w_j F_j$$

Mean Weighted Tardiness $$\overline{T}_w = \sum_{j=1}^{m} w_j T_j$$

Maximum Flowtime $$F_{max} = \max_{1 \le j \le m} \{F_j\}$$

Maximum Tardiness $$T_{max} = \max_{1 \le j \le m} \{T_j\}$$

Number of Tardy Jobs $$N_t = \sum_{i=1}^{m} \delta(T_j)$$

where $\delta(x) = 1$ if $x > 0$

$\delta(x) = 0$ otherwise

In some areas, all jobs have equal importance. This will be called the unweighted case or $w_j = 1$. In this situation, the first two performance measures simplify to the following:

Mean Flowtime $$\overline{F} = \frac{1}{m} \sum_{j=1}^{m} F_j$$

Mean Tardiness $$\overline{T} = \frac{1}{m} \sum_{j=1}^{m} T_j$$

Notice that all the above measures are nondecreasing functions of the completion times. In fact, these measures belong to an important class of measures called regular measures. More precisely, suppose $Z = f(C_1, C_2, \ldots C_m)$ is a measure for schedule S and $Z^1 = f(C_1^1, C_2^1, \ldots C_m^1)$ is the same measure under a different schedule S^1. Then Z is a regular measure if $Z^1 > Z$ implies that $C_j^1 > C_j$ for some job j.

Regular measures are useful because they usually allow one to consider only a restricted subset of all possible schedules when searching for an optimal schedule. This subset is called a dominant set. For example, let D be the dominant set of schedules. Then, for any schedule $S \notin D$ there is a schedule $S' \in D$ which is at least as good as S. Consequently, an optimal solution can always be found in D.

Classification Schemes

In the spirit of queueing theory, several classification schemes have been developed for scheduling problems. The following scheme is a hybrid of several schemes [7] [22].

The classification has the following format:

$$\alpha/\beta,\Gamma/\gamma/\delta$$

where:

α represents the number of processors,

β represents the precedence structure,

Γ represents additional restrictions on the schedule (possibly empty),

γ represents the task execution times,

δ indicates the performance measure.

For example, $2|tree|1|F_{max}$ would represent a scheduling problem for two processors, the tasks having a tree precedence structure and unit execution time, and the performance measure to be optimized being maximum flowtime.

D. SELECTED ALGORITHMS FOR SCHEDULING PROBLEMS

This work is primarily concerned with scheduling problems in which each task requires the same amount of time to complete. This is known as unit execution time. Problems not requiring unit execution time will be briefly considered in this section only to provide a complete overview of the complexity of scheduling problems. Also, of the many possible performance measures, only two will be considered: F_{max} and $\overline{F}_{w}$.

Algorithms for Minimizing F_{max}

T. C. Hu [14] solved the $m|tree|1|F_{max}$ problem. The algorithm is based on a level structure in which tasks are partitioned into levels based on the number of tasks following each task in the unique path to the terminal task. The terminal task is the task that has no successor. Tasks are then scheduled giving priority to the highest levels. Hu's proof of optimality of the algorithm is to find a bound on the number of processors required to complete all tasks in a prescribed length of time and then show that the algorithm achieves this bound. This type of proof is quite common in scheduling theory and will be used in later chapters of this work. Actually, Hu's algorithm can be extended to a forest precedence structure by observing that if the algorithm does not

provide the optimal schedule for a forest, then the algorithm cannot provide the optimal schedule for a tree precedence structure derived by adding a new task that is a successor to all terminal tasks in the forest.

Muntz and Coffman [21] extended Hu's algorithm for the case of arbitrary execution times. However the tasks must be preemptable.

Another special case for which an optimal algorithm exists is due to Coffman and Graham [7]. They add the restriction that there can be only two processors; however, the precedence structure may be general. The algorithm uses the same labeling scheme as Hu's algorithm although the arbitrary precedence structure introduces some complication since there is no longer a unique path to a terminal task. The labels now represent the sum of the execution times in the longest execution time path to a terminal task. In the case of unit execution time, the labels are the number of tasks in the longest path to a terminal task. An example of unit execution time arbitrary precedence structure with labels above each node is given in Figure 2.3.

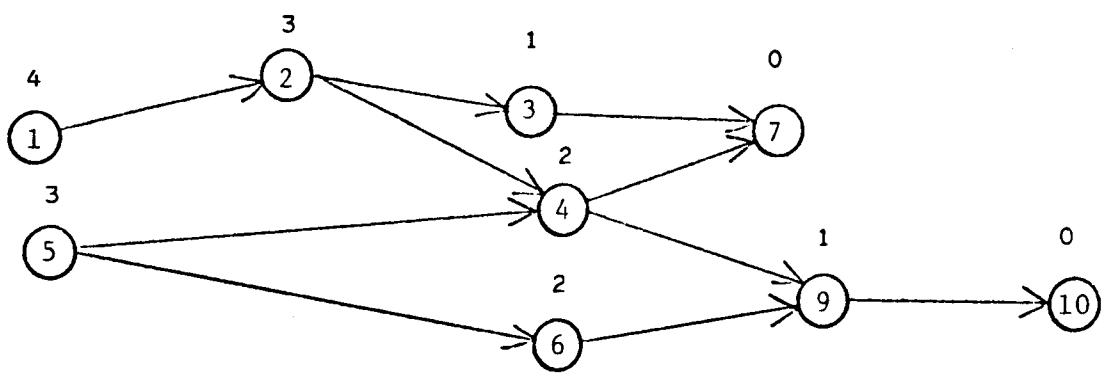

Figure 2.3. Arbitrary Task Structure.

The scheduling phase of this algorithm uses list scheduling with jobs having the largest labels at the top of the list. List scheduling means that when a processor becomes available, the list is scanned from the top and the first ready job encountered is scheduled. Although not mentioned in the original article, the algorithm may not produce an optimal list if the precedence structure contains transitive arcs. In Figure 2.4, arc (a,c) is a transitive arc and must be deleted if the algorithm is to work correctly.

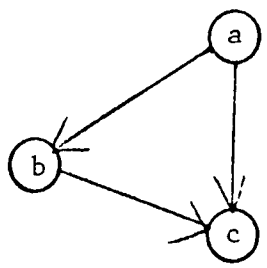

Figure 2.4. Example of a Transitive Arc.

Aho, Garey, and Ullman [2] developed an algorithm to delete transitive arcs in at worst $O(n^{2.8})$ time. Therefore the problem of transitive arcs causes no real complexity problem.

For more than two processors neither the Coffman-Graham algorithm nor any other known algorithm provides an optimal solution in polynomial time. The complexity of this problem with the number of processors fixed and greater than two still remains an open question. However if the number of processors is a free parameter Ullman [25] has shown the problem to be NP-Complete.

The complexities of problems minimizing F_{max} are summarized in Table 2.1.

Table 2.1. Complexity Results for F_{max}

Number of Processors	Execution Times	Precedence Structure	Problem Complexity
-	Equal	Forest	$O(n)$
-	-	Forest	$O(n\log n)$
2	-	-	$o(n^2)$
Fixed $\geq$ 3	Equal	-	open
-	Equal	-	NP-Complete

Algorithms for Minimizing $\overline{F}_w$

Lawler [18] showed that even when all task weights and task lengths are equal to one and there is only one processor, the $1|\text{arbitrary}|1|\overline{F}$ is NP-complete. There are however several special cases for which polynomial algorithms exist.

If there is no precedence structure, Smith [24] showed that $\overline{F}_w$ can be minimized by executing jobs in increasing order of e_j/w_j. Consequently the $1|\text{empty}|\text{arbitrary}|\overline{F}_w$ problem can be solved in $O(n \log n)$ time. When the precedence structure is a forest, Horn [12] was able to solve the $1|\text{forest}|\text{arbitrary}|\overline{F}_w$ problem in $O(n^2)$ time.

In the case of a fixed number of multiple processors, the problem can be solved in pseudopolynomial time [4]. However if the number of processors is arbitrary, then the problem is NP-complete in the strong sense [19]. In the case of equal weights and an empty precedence structure, Horn [13] solved the $m|\text{empty}|\text{arbitrary}|\overline{F}$ in $O(n^3)$ time. When a precedence structure is added, Sethi [23] has shown that even the $2|\text{tree}|\text{arbitrary}|\overline{F}$ is NP-complete.

The complexities for problems which minimize $\overline{F}_w$ is summarized in Table 2.2.

Table 2.2. Complexity Results for $\overline{F}_w$

Number of Processors	Execution Times	Weights	Precedence Structure	Problem Complexity
1	Equal	Equal	-	NP-complete
1	-	-	Empty	$O(n \log n)$
1	-	-	Forest	$O(n^2)$
-	-	Equal	Empty	$O(n^3)$
2	-	Equal	Tree	NP-complete
2	-	-	Empty	NP-complete

CHAPTER 3

OVERVIEW OF THE SIMULATION

A. SITUATIONS THAT CAUSE DELAYS

Most students enrolled in academic institutions have graduation as their goal and, subject to certain constraints, they would like to graduate as quickly as possible. However, in pursuing their degrees, many students encounter problems in scheduling courses which causes them not to graduate as soon as they would like. For example, if a program is designed to be six quarters long and a student requires seven quarters to graduate, that student encounters a delay of one quarter.

The general problem then is to schedule courses to quarters such that student delay is minimized. There are many reasons for a student to encounter delay.

1. Not enough information is available to the student to make a rational schedule.

Since a student's schedule must be planned in advance, if the student does not know in which future time periods each course is going to be offered, it would be impossible to determine a reasonable schedule. This information, however, is generally available; and the only problem arises when the student fails to acquire it.

2. Students create bad schedules.

Given that the student has perfect information about the future schedule of courses, it is still possible for a student to make up a

schedule that does not enable him to graduate as quickly as possible. Again, this is not a major problem because most institutions provide the student with a suggested course schedule. However, most course schedules are designed for students starting in the fall; and if a student enters in an off quarter, say winter, such suggested schedules will be less useful.

3. Students encounter time conflicts in a particular quarter.

If a student needs to take two courses in the same quarter and both of which are offered at the same time, significant delays might result because the remainder of the student's schedule was developed assuming that these two courses would be completed in that quarter.

4. Students need to take a class that is already full.

Again, significant delays can result because the predetermined schedule of the student can no longer be followed.

Most ongoing programs have indeed minimized all the previously mentioned problems by a rather painful trial and error process. The model in this work is designed to eliminate or at least reduce the trial and error process in the actual institution by simulating the process.

B. GENERAL ASSUMPTIONS OF THE SIMULATION MODEL

In designing the simulation model, perfect information was assumed available to the student. Consequently, no student will incur delays because of lack of information. Also, the scheduling algorithm designs good schedules for the students based on the information available. These

schedules are not guaranteed to be optimal, however. Details of the algorithm used for an individual student to derive his schedule are given in Chapter 4.

The present simulation model does not include the delays students might encounter due to courses being offered at the same time in a particular quarter. In other words, if a student plans to take courses A and B in a certain quarter and both classes are not full, then that student will always be able to do so. The model does include, however, a report called a conflict matrix that includes the number of students taking both A and B in the same quarter.

Also, in the simulation model class size limits are strictly enforced. If the class size is set at 40, under no circumstances will the model allow 41 students to be enrolled.

C. INPUTS TO THE SIMULATION MODEL

Perhaps the best way to understand the situations the simulation model is capable of handling is to consider the type of data required as input to the model. This information has been divided into files. The exact input format of the file is given in the Appendix, but an overview of the data is presented here.

Course File

The course file contains a list of all courses and the maximum number of students allowed in each section. In addition, it contains all the prerequisites and co-requisites for each course and the department responsible for staffing the course. In this way, it is

possible for a course listed under a department to be staffed by another department or a pseudodepartment. A pseudodepartment is any subset of the faculty that is not an academic department but which staffs a group of courses. For example, in accounting, most auditing courses would be staffed by a subset of the accounting department staff specializing in auditing. Auditing could, therefore, be made a pseudodepartment and all auditing courses assigned to it.

Department File

The department file contains the number of sections each department or pseudodepartment can staff in each quarter.

Curriculum File

The curriculum file contains a list of all courses required for each curriculum. It also contains the maximum number of courses a student following each curriculum can take per quarter. In this way, a distinction between part-time and full-time students can be made. The file also contains an optional list of those courses that can be substituted for each course in the curriculum if the original course cannot be scheduled. It is also possible to list the courses for which a student following a particular curriculum has been given credit. In this way, transfer students or graduate students with differing undergraduate degrees can be handled.

Also included are the probabilities that a student will drop out of school. These probabilities may vary depending on the number of quarters

a student has been enrolled, thus allowing for the fact that the drop out rates are typically higher in the early quarters of a student's program.

Projected Enrollment File

The projected enrollment file contains the number of students expected to enter each curriculum in each quarter.

Initial Schedule File

The initial schedule file contains the number of sections of each course that will be offered in each quarter. This file is optional and is used to evaluate a current schedule or a proposed new schedule.

D. THE PROCESSOR OF THE SIMULATION MODEL

The simulation processor can be divided into three modules:

1. Generation of initial schedule
2. Simulation based on current schedule
3. Modification of current schedule

Generation of Initial Schedule

The initial schedule can be generated in one of two ways. The first way is to offer every course every quarter with infinite course capacities. Essentially, this type of initial schedule will enable all students to enroll in every course exactly when they want to take it.

The second method is to use the schedule found on the Initial Schedule File as the starting schedule.

Either of these two methods of initialization can be chosen depending on the simulation mode used.

Simulation Based on Current Schedule

Given that a schedule exists, the simulation module attempts to schedule each student in such a way that the time until graduation is minimized. Each student is simulated individually. Consequently, two students following exactly the same curriculum may have different schedules due to one of the students encountering full classes. Since the simulation is for 20 quarters, the order in which students are processed is as follows. All students who enter in period 1 are scheduled first, then all students starting in period 2 and so on. Within each period, one student is processed from each curriculum, then a second student from each curriculum and so on until all students have been simulated in that period. The exact algorithm by which students schedule classes is given in the next chapter.

Modification of Current Schedule

After all students have been simulated, it may be necessary to make some course changes. This can be done in two ways: manually or automatically. In manual mode, the operator may delete, add or transfer a section of any course. In automatic mode, the system deletes a section by a predetermined algorithm, given in the next chapter, in an attempt to make all departments feasible. A feasible department is one in which all courses can be staffed by the faculty available in each quarter.

E. REPORTS GENERATED BY THE SIMULATION MODEL

There are several different reports that can be requested by the operator. A brief description and example of each report follows.

Enrollment Report

The enrollment report gives the number of students enrolled in each class in each quarter. An example of this report is given in Table 3.1.

Table 3.1. Enrollment Report

	Fall	Winter	Spring	Summer
AC5010	116	0	0	39
AC5020	153	47	0	0
AC5030	0	134	49	0
BL5010	60	0	0	53
EC5010	116	0	0	39
EC5020	145	114	0	0
EC5030	0	0	117	59
FN5010	0	135	33	0
FN5020	0	0	78	73
MG5010	52	41	0	101
MG5020	57	0	55	61
MS5010	0	71	123	19
MK5010	68	63	64	0
MK5020	5	0	69	90
ST5010	135	126	0	0
ST5020	0	99	117	0
BA5310	59	27	0	57

Staffing Report

The staffing report gives the number of sections that each department must staff in each quarter under the current schedule. An example of this report is found in Table 3.2.

Table 3.2. Department Staffing Report

	Fall	Winter	Spring	Summer
Accounting	10	6	2	2
Business Law	2	0	0	2
Economics	9	4	0	2
Finance	0	5	5	3
Management	4	2	2	4
Management Science	0	3	5	1
Marketing	4	3	6	3
Statistics	5	9	4	0
Business Admin.	2	1	0	2

Conflict Matrix

To aid in timetabling courses within each quarter, a conflict matrix can be generated. An example of this report is given in Table 3.3. This report shows how many students are taking two courses in the same quarter. For example, 57 students are taking EC5010 and AC5010 in the fall quarter, and care should be taken that these two courses are not offered at the same time since the schedule of 57 students would be affected.

Table 3.3. Conflict Matrix for Fall Quarter

	BA5301	ST5010	MK5010	MK5020	MG5010	MG5020	EC5010	EC5020	BL5010	AC5020
AC5010	0	0	0	0	0	17	0	57	38	0
AC5020	0	38	0	37	28	17	43	0	0	
BL5010	0	3	0	0	0	17	0	38		
EC5010	0	0	0	0	0	17	0			
EC5020	0	53	0	22	0	12				
MG5010	0	17	0	0	0					
MG5020	0	0	0	0						
MK5010	0	8	0							
MK5020	0	0								
ST5010	0									

Delay Report

The delay report gives the number of extra quarters required for students following the different curricula to graduate. An example of this report is given in Table 3.4. For those students following curriculum MBA 3, which stands for taking three courses per quarter, seven students are delayed one quarter and fifteen students are delayed two quarters. Notice that for those students taking four quarters per quarter, MBA 4, there is no delay since the program was designed for this student load.

Table 3.4. Lateness Report

	Lateness (in Periods)							
Curriculum	0	1	2	3	4	5	6	7
MBA 4	22	0	0	0	0	0	0	0
MBA 3	0	7	15	0	0	0	0	0
MBA 2	18	13	0	0	0	0	0	0
MBA 4 (B.U.)	0	24	0	0	0	0	0	0
MBA 3 (B.U.)	0	12	4	0	0	0	0	0
MBA 2 (B.U.)	0	26	0	0	0	0	0	0

Individual Student Report

The individual student report gives the schedules for individual students. It is useful for determining exactly where students experience problems in their course schedules. An example is given in Table 3.5.

These three modules can be used in the predetermined simulation modes: Manual, Automatic, and Evaluation.

Table 3.5. Individual Student Report

MBA 4				
Fall	AC5010	BL5010	EC5010	MG5010
Winter	AC5020	ST5010	EC5020	MK5010
Spring	ST5020	MS5010	AC5030	FN5010
Summer	EC5030	FN5020	MG5020	MK5020
Fall	BA4310			
MBA 3				
Fall	AC5010	BL5010	EC5010	
Winter	MG5010	AC5020	ST5010	
Spring	ST5020	MS5010	AC5030	
Dropout				
MBA 2				
Fall	AC5010	EC5010		
Winter	ST5010	EC5020		
Spring	ST5020	MS5010		
Dropout				
MBA 4 (B.U.)				
Fall	AC5020	ST5010	EC5020	MG5010
Winter	AC5030	FN5010	MK5010	ST5020
Spring	EC5030	MS5010	FN5020	MK5020
Summer	MG5020			
Fall	BA5310			
MBA 3 (B.U.)				
Fall	AC5020	ST5010	EC5020	
Winter	MG5010	AC5030	FN5010	
Spring	MK5010	ST5020	EC5030	
Summer	MS5010	FN5020	MK5020	
Dropout				
MBA 2 (B.U.)				
Fall	ST5010	EC5020		
Winter	ST5020	MS5010		
Spring	MK5010	EC5030		
Summer	MG5010	MK5020		
Fall	AC5020	MG5020		
Winter	AC5030	FN5010		
Spring	FN5020			
Summer	BA5310			

Table 3.5 (continued)

MBA 4				
Fall	AC5010	BL5010	EC5010	MG5010
Winter	AC5020	ST5010	EC5020	MK5010
Spring	ST5020	MS5010	AC5030	FN5010
Summer	EC5030	FN5020	MG5020	MK5020
Fall	BA5310			

Manual Mode

In manual mode, the initial course schedule is generated such that every course is offered every quarter with infinite capacities. The simulation module is then executed so that reports can be generated enabling the operator to decide if there should be any course changes. After all the schedule changes have been made manually, the simulation module can be executed again. This process is depicted in a flowchart given in Figure 3.1.

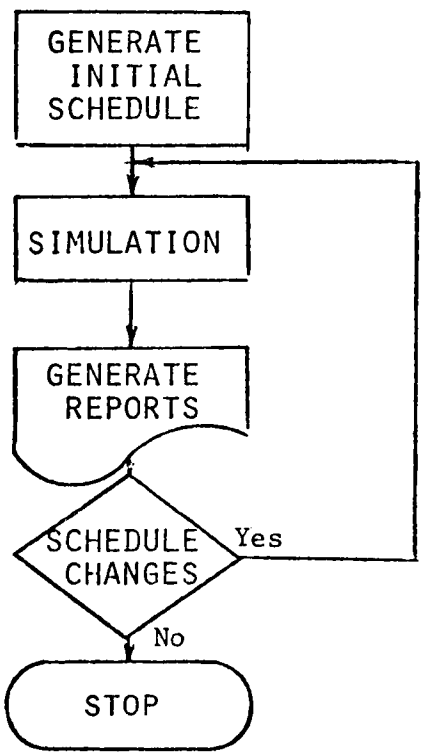

Figure 3.1. Flowchart for Manual Mode.

Automatic Mode

In automatic mode, the initial course schedule is generated such that every course is offered every quarter with infinite capacities. The simulation module is then executed, after which courses are deleted

automatically in an attempt to achieve a feasible schedule. The exact algorithm used for the deletions is presented in the next chapter. A summary of automatic mode is given in Figure 3.2.

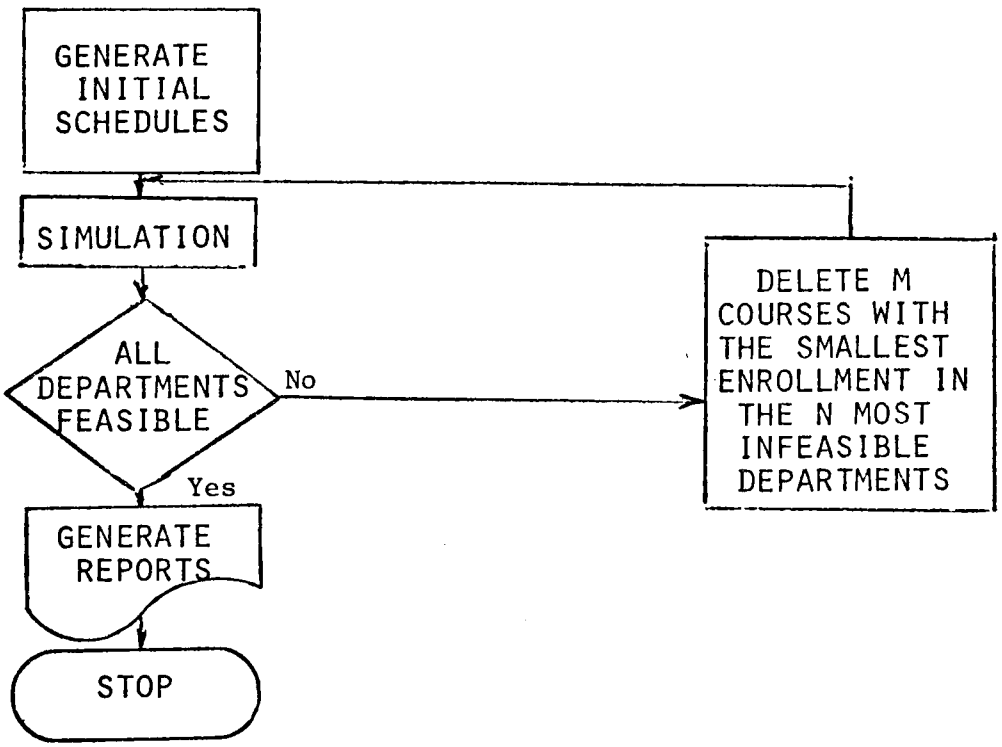

Figure 3.2. Flowchart for Automatic Mode.

Evaluation Mode

In evaluation mode, the initial course schedule is read from the Initial Schedule file. The simulation module is then executed. The operator may then make manual changes to the course schedule. This process is shown in Figure 3.3.

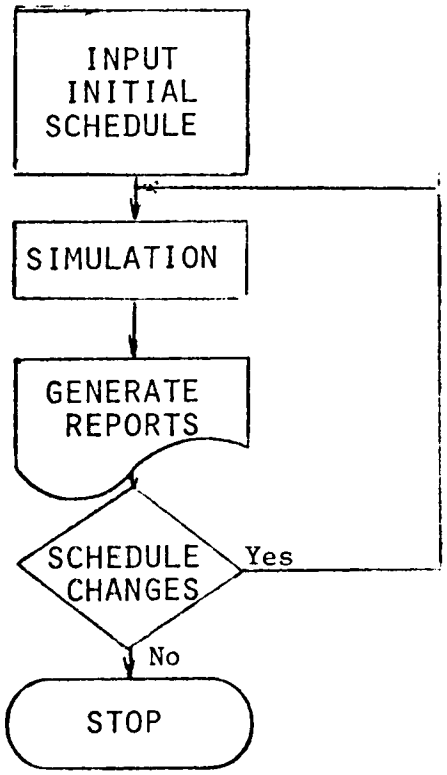

Figure 3.3. Flowchart for Evaluation Mode.

The simulation is run for twenty quarters. However, only quarters nine, ten, eleven, and twelve are used for reports and statistics to eliminate the startup and shutdown effect of the simulation. For example, if a student starts taking classes in the first quarter of the simulation, he will encounter no full classes caused by students who entered in a previous quarter. To eliminate this startup bias, the first eight quarters of the simulation are used only to achieve a type of steady state situation. Similarly, the last eight quarters of the simulation are omitted because a student entering in one of these periods may not have time to graduate before the twentieth period is reached and the

simulation terminated. The model uses a cycle length of four quarters which means that the same set of courses will be offered every fourth quarter. Both the number of quarters for the simulation and cycle length can be easily changed at system generation time.

CHAPTER 4

RUNNING THE SIMULATION SYSTEM

A. INTRODUCTION

The simulation system is written in FORTRAN designed to run on a DEC-System-10 or with slight modifications on any interactive FORTRAN system.

The system is designed to be used by a nontechnical operator required to learn only a minimal number of commands. Whenever operator action is requested, the system will prompt the operator with an appropriate message. If at any time the operator is not sure of the proper response, the word HELP may be typed. This will list the possible responses and give a brief explanation of each.

B. SIMULATION MODE SELECTION

The simulation system operates in three modes: Manual, Automatic, and Evaluation. The operator must initially choose one of these three modes. The first prompt gives the operator this opportunity.

ENTER SIMULATION MODE →

The operator should then respond with the desired mode. Only the first character of each mode need be typed.

In addition to choosing the simulation mode, various switches may be set in the same response. The complete form of the response is

ENTER SIMULATION MODE → simulation mode/switch 1/switch 2/...

where each switch is separated by a slash. The possible switches are REPORT, DEBUG, and FULL. None, all, or any combination can be specified. Only the first character of each switch is required.

For example, if evaluation mode with the REPORT and DEBUG options is desired, the following command should be entered:

ENTER SIMULATION MODE → E/D/R

which is equivalent to

ENTER SIMULATION MODE → EVALUATION/DEBUG/REPORT

The simulation switches control the amount of output generated as the simulation system executes. A brief description of each switch follows.

DEBUG

The DEBUG option requests that individual students' programs be traced through the simulation. The option will generate course schedules for five students in each curriculum in each of the four quarters. An example is given in Table 3.5, p. 28.

REPORT

The REPORT option will automatically call for the generation of the Course Report, Departmental Report, and Lateness Report after each execution of the simulation module. Samples of these reports are given in Chapter 3.

FULL

The FULL option causes all reports to include statistics for all the time periods in the simulation. Normally, only periods 9, 10, 11, and 12 are used for reports to eliminate problems with the simulation initiation

and termination. If the FULL option is specified, all twenty periods of the simulation will appear in the reports.

If automatic mode is selected, the automatic course deletion module requires further inputs. The system will prompt the operator with:

NUMBER OF COURSES TO BE DELETED PER DEPARTMENT →

and

NUMBER OF DEPARTMENTS TO CONSIDER FOR DELETIONS →

To both of these prompts, the operator should respond with a positive integer. Details of the automatic deletion module are included later in this chapter.

C. REPORT SELECTION

After the simulation mode has been specified and the simulation module executed, the operator may then ask for various reports. This is indicated by the prompt

ENTER ACTION →

The syntax for the response to this prompt depends on the report desired.

Course Enrollment Report

The entire Course Enrollment Report can be requested by entering

ENTER ACTION → C

It is also possible to ask for enrollment reports for any particular department or course. In asking for a department report, the C must be followed by a comma and then the two-letter department code. For example,

ENTER ACTION → C,AC

would generate the course enrollment report for all courses in the Accounting Department. To request the enrollment for a particular course, the C must be followed by a comma and then by the six-character course code. For example,

ENTER ACTION → C,AC5010

would give the enrollment report for Accounting 5010.

Lateness Report

The summary lateness report for all students can be requested by the command

ENTER ACTION → L

It is also possible to break down the lateness by each curriculum giving a separate report for each curriculum. This can be accomplished by entering

ENTER ACTION → L,CU

Department Report

To generate the department staffing report, the operator must enter the following command

ENTER ACTION → D

If the staffing report is desired only for a particular department, then the D must be followed by a comma and the two-character department code. For example,

ENTER ACTION → D,MK

would generate the staffing report for the Marketing Department.

Conflict Matrix

The conflict matrix can be requested by entering the following command

ENTER ACTION → M

In many cases, this report is so large that it will not fit on a terminal with an 80-column screen. By entering the command

ENTER ACTION → M,L

the conflict matrix will be printed on the line printer.

D. COURSE SCHEDULE CHANGE SELECTION

In addition to requesting the various reports, the operator may make schedule changes. The general form of the commands that request course schedule changes is

ENTER ACTION → action code, course code

where the action codes are

A - add a section

K - kill a section

T - transfer a section

For example, the command

ENTER ACTION → K,AC5010

would request that a section of Accounting 5010 would be killed. To both the A and K commands, the system will respond

PERIODS →

The operator should then indicate in which period or periods the addition or deletion should take place. The period codes are as follows:

1 - Fall

2 - Winter

3 - Spring

4 - Summer

More than one period may be specified. For example, if the operator entered the following commands

ENTER ACTION → K,AC5010

PERIODS → 1,2,2,4

one section of Accounting 5010 would be deleted in the fall quarter, two sections deleted in the winter quarter, and one section deleted in the summer quarter.

If a deletion request is made for a course that is not offered in a period, no action is taken.

If transferring a section is desired, the system prompts are slightly different as can be seen in the following example.

ENTER ACTION → T,ST5010

FROM PERIOD → 1

TO PERIOD → 2

In this example, a section of Statistics 5010 will be transferred from the fall quarter to the winter quarter. In the transfer command, only one period can be specified after the FROM PERIOD and TO PERIOD prompts. Any request for a course change takes place cyclically. For example, if a course is added in the fall quarter, it is added in all fall quarters over which the simulation is run.

E. RE-EXECUTING OR TERMINATING THE SIMULATION

After all desired changes have been made, the simulation module can be executed by entering a blank command after the prompt ENTER ACTION →.

Reports will not reflect any of the course changes until the simulation module is re-executed.

The operator can terminate the entire simulation system by entering

ENTER ACTION → STOP

F. PROGRAMMING DETAILS OF THE SIMULATION MODULE

The problem facing each individual student is to determine a course schedule that will enable that student to graduate as quickly as possible. In determining this schedule, the prerequisite and co-requisite course structure must be followed; also, since not all courses are offered every quarter, the list of quarters in which courses are offered must be considered.

This problem falls in a category of scheduling problem known as precedence constrained scheduling with resource constraints. As was demonstrated in Chapter 2, this problem is in general NP-Complete. Therefore, there are two alternatives in "solving" the students' problem:

1. Develop a new heuristic specifically designed for the students' problem.

2. Use an enumeration technique, such as branch and bound, specifically designed for the students' problem.

Because of the number of students that must be simulated to give reasonable results, the second alternative was considered to be too time consuming. Consequently, the heuristic approach was used.

As was mentioned in Chapter 2, the optimal algorithms for special cases of precedence constrained scheduling problem use a type of labeling method. Most heuristics use similar labeling techniques which in general produce good results.

The heuristic designed to solve the students' problem is a combination of a labeling scheme and list scheduling. In developing the labels, there are several factors that must be considered. The label must take into account the frequency that a course is offered. For example, if a particular course is offered only once per year, then a student who has all the prerequisites satisfied should take this course before a course that is offered every quarter.

The label must also consider the critical path of the prerequisite structure. For example, consider the prerequisite structure shown in Figure 4.1.

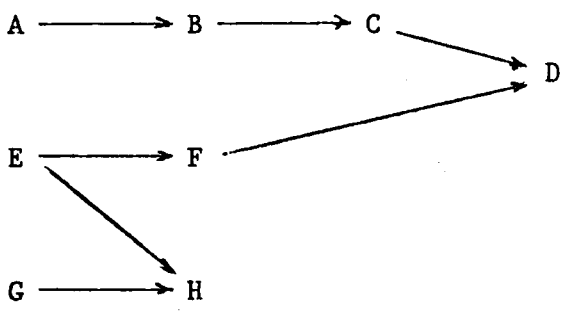

Figure 4.1. Sample Prerequisite Structure.

The minimum number of time periods required for a student to complete all the courses, regardless of the number of courses taken per period, is four. This is because there is the chain A → B → C → D, which has length four. The longest chain is called the critical path, and its length provides a lower bound on the number of time periods required to complete all the courses. Consequently, if a student has a choice between taking courses A, E or G, course A is in a sense the most critical since it is on the critical path.

To incorporate this concept into the labeling process, each course is given a label which represents the number of courses following it in the longest chain to a terminal course. A terminal is a course that has no course following it in the prerequisite structure. The previous example would have labels as shown in Figure 4.2. Notice that all terminal courses are given a label of zero.

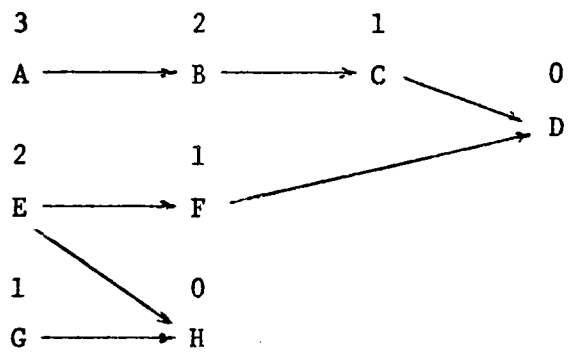

Figure 4.2. Sample Prerequisite Structure with Labels.

The labeling scheme used to solve the students' problem uses a combination of the frequency that a course is offered and the longest following chain label.

Heuristic for Students' Problem

Labeling Phase

1. Assign a weight w_j to each course j.

$$w_j = N - n_j + 1 \tag{4.1}$$

where N = the number of time periods in the cycle,

n_j = the number of times course j is offered in the N periods of the cycle.

2. Assign the label w_j to each terminal course j.
3. Assign to course j the label α_j.

$$\alpha_j = \max \{\alpha_i \mid_i << _j\} + w_j \tag{4.2}$$

where << stands for immediate predecessor.

4. Order the courses in a list in decreasing order of α.

Scheduling Phase

5. Set i = period in which the student begins the program.
6. Set c = the course at the top of the list.
7. IF course c is offered in period i and all prerequisites have been satisfied

   THEN

   schedule course c in period i,

   delete course c from the list.
8. IF all courses have been scheduled

   THEN STOP.

9. IF m courses have been scheduled in period i

   THEN

   set i + i + 1

   GO TO 6

   ELSE

   set c to the next course in the list

10. IF the end of the list has been reached

    THEN

    set i = i + 1

    GO TO 6

    ELSE

    GO TO 7

Consider the following example problem where the course schedule is given in Figure 4.3.

	PERIOD				
COURSE	1	2	3	4	w_j
A	X		X		3
B		X		X	3
C	X		X		3
D	X				4
E		X			4
F			X		4
G	X		X		3
H	X				4
I		X		X	3
J			X		4
K	X			X	3
L	X	X	X		2
M		X	X	X	2
N		X		X	3

Figure 4.3. Schedule of Course Offerings.

If a cycle length of 4 is assumed, then using step 1 in the labeling phase, weights w_j can be calculated. These are given in the last column of Figure 4.3.

To graduate, a student must take all of the courses. The precedence structure is given in Figure 4.4.

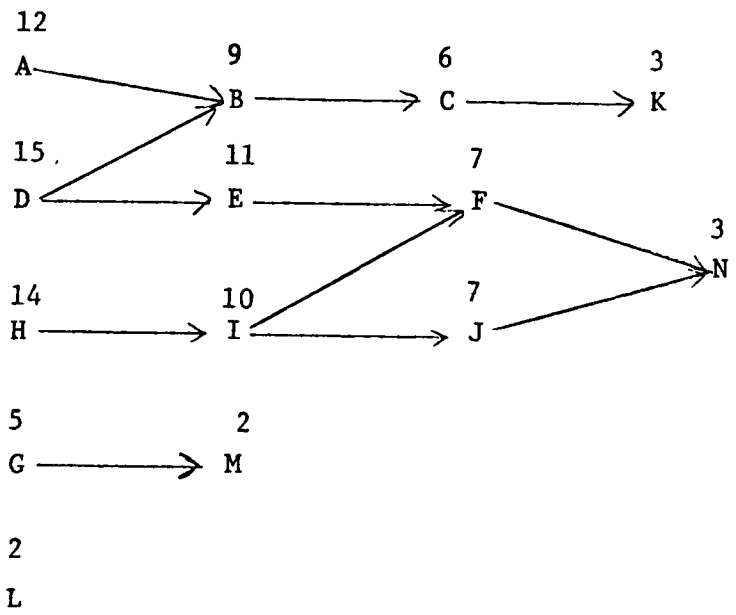

Figure 4.4. Example Course Precedence Structure.

The numbers above each course represents the labels that would be generated using the labeling phase of the algorithm. Putting these courses in decreasing label order would generate the following priority list.

D H A E I B F J C G K N M L

The scheduling phase of the algorithm can then be applied to the course list. For example, suppose a student can take a maximum of two

courses per quarter and that student entered in quarter one. The student's schedule as generated by the heuristic is given in Figure 4.5.

QUARTER	1	2	3	4	5	6	7	8
COURSES TAKEN	D	E	A	B	C	M	J	K
	H	I	F		G	L		N

Figure 4.5. Student's Schedule for Two Courses per Quarter.

Similarly, if a student can take three courses per quarter, then that student's schedule is given in Figure 4.6. Both of the two

QUARTER	1	2	3	4	5	6
COURSES TAKEN	A	B	C	K	G	M
	D	E	F	N	L	
	H	I	J			

Figure 4.6. Student's Schedule for Three Courses per Quarter.

previous schedules generated by the heuristic turn out to be optimal. Certainly, optimality is not guaranteed by the heuristic; however, in all but some contrived pathological cases, the heuristic does quite well. If such cases do occur, however, it is possible to bypass the labeling phase of the algorithm by directly inputting the priority list to be used by the scheduling phase of the algorithm. This can be done by using the NOSORT statement in the curriculum file.

G. PROGRAMMING DETAILS OF THE AUTOMATIC DELETION MODULE

If the user specifies automatic mode, the system will attempt to make all departments feasible in all quarters. By department feasibility

is meant that the number of sections offered per quarter does not exceed the number that can be offered based on the department input file.

Since in automatic mode the initial schedule is developed without regard to staffing constraints, this schedule is usually infeasible. In order to gain feasibility, course sections must be deleted. The simulation system uses the following algorithm in determining which courses to delete.

Let c_{ij} = maximum enrollment in each section of course i offered by department j.

Let e_{ijk} = current enrollment in each course i offered by department j in period k.

s_{kj} = maximum number of sections department j can offer in period k.

Step 1 Determine n_{kj}, the number of sections offered by department j in period k

$$n_{kj} = \sum_{i} \left\langle \frac{e_{ijk}}{c_{ij}} \right\rangle$$

where $\langle x \rangle$ denotes the smallest integer greater than or equal to x.

Step 2 Determine I_{kj}, the infeasibility of department j in period k

$$I_{kj} = \max \{0, n_{kj} - s_{kj}\}$$

Step 3 Determine I_j, the total infeasibility of department j over some set of periods K

$$I_j = \sum_{k \epsilon K} I_{kj}$$

In the current version of the simulation, twenty quarters are simulated but K = {9, 10, 11, 12} to eliminate initiation and termination effects of the simulation.

Step 4 Determine j*, the most infeasible department

$$j^* = \{j \mid \max I_j, \text{ for all } j\}$$

Step 5 Determine k*, the most infeasible period for department j*

$$k^* = \{k \mid \max I_{kj^*}, k \epsilon K\}$$

Step 6 Determine i*, the course with the smallest enrollment in a section in period k* for department j*

$$i^* = \{i \mid \min [e_{ij^*k^*}]_{c_{ij^*}}, \text{ for all } i\}$$

where $[c]_b$ is c modulo b

Step 7 Delete course i* offered in period k* offered by department j*

If just one section is deleted between each simulation module execution, the number of simulation module executions required until feasibility is achieved would be large. Consequently, it is useful to delete more than one section between simulation module executions.

If automatic mode has been specified, the user will be given the following two prompts:

NUMBER OF COURSES TO BE DELETED PER DEPARTMENT →

NUMBER OF DEPARTMENTS TO CONSIDER FOR DELETIONS →

Let m be the response to the first prompt and n the response to the second prompt. The deletion algorithm will then be executed as follows:

Step 1

Step 2

Step 3

Step 4

Step 5 ⎫ repeated ⎧ repeated
Step 6 ⎬ m times ⎨ n times
Step 7 ⎭ ⎩

In automatic mode, the simulation module and the automatic deletion mode will be alternately executed as many times as necessary until all departments are feasible.

CHAPTER 5

RESULTS OF THE SIMULATION MODEL

A. INTRODUCTION

The simulation model was run using data from the MBA Program at The University of Tennessee. At the time the model was run, there was a considerable lack of accurate data. After conferring with the program administrators, estimates were made for the various data items. The model was used only for the core courses in the MBA Program. As more data becomes available concerning enrollment in the various concentrations, the model could be expanded to include all of the courses in the MBA Program.

B. INPUT DATA FOR THE SIMULATION MODEL

There are seventeen core courses in the MBA Program at The University of Tennessee. The prerequisite and co-requisite structure is depicted in Figure 5.1. The small "c" above certain arcs represents a co-requisite relation.

Students entering the MBA Program can be divided into two major categories—those that have business undergraduate degrees and those with nonbusiness undergraduate degrees. The major distinction between the two types is that those students with business undergraduate degrees usually need not take Accounting 5010, Economics 5010, and Business Law 5010. Obviously, some students do not fit exactly into

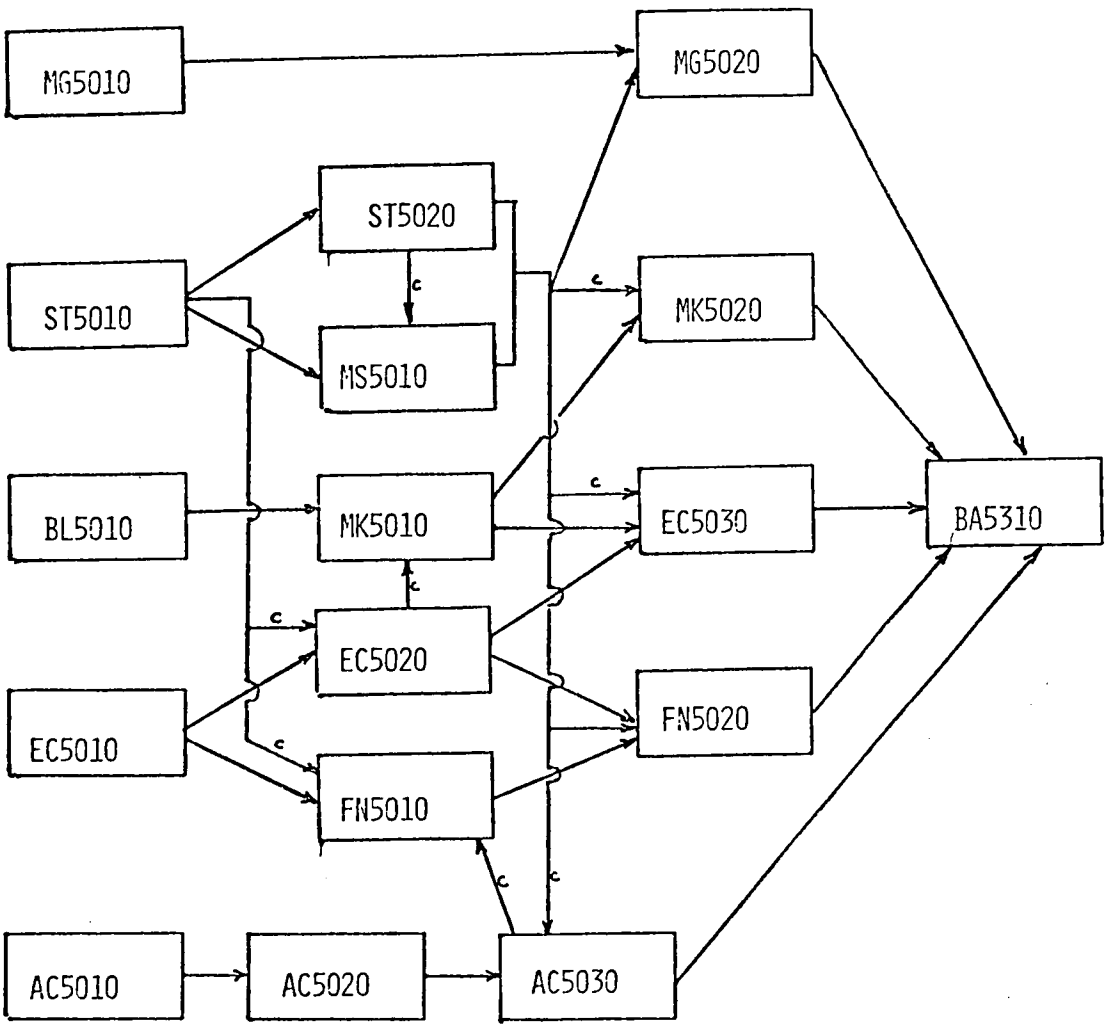

Figure 5.1. Sequence of MBA Core Courses, The University of Tennessee.

either category, but because of their relatively small numbers, they are ignored.

These two categories can be further divided depending on the maximum number of courses each student can take per quarter. Most students take either two, three, or four courses per quarter. Consequently, there were six distinct classifications that were used in the simulation model. These are listed below.

MBA 4

MBA 3

MBA 2

MBA 4 (B.U.)

MBA 3 (B.U.)

MBA 2 (B.U.)

The number indicates the maximum number of courses that a student can take per quarter. Students with undergraduate degrees in business are indicated by "B.U."

The MBA Program at Tennessee is a lockstep-type program which forces students to enter in certain periods. For example, those students with business undergraduate degrees should enter in either the fall or winter quarters, while those students with nonbusiness undergraduate degrees should enter in either the summer or fall quarters.

After considerable effort, the approximate number of students entering in each of the six classifications in each quarter was determined. These numbers are given in Table 5.1.

Table 5.1. New Admissions Broken Down by Classification and Quarter

Classification	Quarter			
	Fall	Winter	Spring	Summer
MBA 4	23	0	0	8
MBA 3	23	0	0	8
MBA 2	70	0	0	23
MBA 4 (B.U.)	21	4	0	0
MBA 3 (B.U.)	21	4	0	0
MBA 2 (B.U.)	63	21	0	0

After reviewing many students' records, it was found that a significant number of students drop out of the program. This seemed to be more prevalent for the part-time students who were taking only two courses per quarter. The reasons for this higher dropout rate are quite obvious when one considers that these students usually work full time. Consequently, many of these students are either transferred or find it too difficult to go to school while maintaining a full-time job. Also, the dropout rates for all students appear to be higher in the earlier quarters of a student's program. The dropout rates that were used in the simulation model are given in Table 5.2.

Table 5.2. Probability of Dropout Broken Down by Classification and Quarters in Program

Classification	Quarter in Program				
	1	2	3	4	5
MBA 4	.10	.09	.08	.07	.05
MBA 3	.10	.09	.08	.07	.05
MBA 2	.20	.18	.16	.14	.10
MBA 4 (B.U.)	.10	.09	.08	.07	.05
MBA 3 (B.U.)	.10	.09	.08	.07	.05
MBA 2 (B.U.)	.20	.18	.16	.14	.10

Only one more input is required for the simulation. If the simulation is run in evaluation mode, an existing course schedule must be input. Table 5.3 gives the course schedule that was used.

Table 5.3. Course Schedule for Core MBA Courses

	Quarter			
Course	Fall	Winter	Spring	Summer
AC5010	X			X
AC5020	X	X		
AC5030		X	X	
BL5010	X			X
BA5310	X	X		X
EC5010	X			X
EC5020	X	X		
EC5030	X		X	X
FN5010		X	X	
FN5020	X		X	X
MG5010	X	X		X
MG5020	X		X	X
MK5010	X	X	X	
MK5020	X		X	X
ST5010	X	X		
ST5020		X	X	
MS5010		X	X	X

C. SIMULATION RESULTS

The simulation was run in both manual and evaluation mode. In both modes, infinite course capacities were used. This was done to help estimate enrollments in various courses. The results for the manual run, where no initial course schedule was provided, are given in Table 5.4.

Table 5.4. Enrollment Figures for Simulation in Manual Mode

Course	Quarter			
	Fall	Winter	Spring	Summer
AC5010	116	0	0	39
AC5020	92	51	39	26
AC5030	21	75	32	44
BL5010	62	56	0	16
EC5010	116	0	0	39
EC5020	113	56	28	36
EC5030	47	21	66	32
FN5010	0	68	51	37
FN5020	43	0	47	62
MG5010	51	44	54	58
MG5020	75	45	7	43
MS5010	3	75	105	33
MK5010	54	46	35	45
MK5020	10	55	61	39
ST5010	136	128	0	0
ST5020	0	100	117	3
BA5310	42	44	24	29

These enrollment figures represent schedules in which each student incurs zero delay since all courses are offered every quarter with infinite capacities. They should provide a good basis from which to develop the actual course schedule to be used. The course schedule decided upon by the administrators was given in Table 5.3. A comparison between the projected demands and the actual course schedule shows that the schedule is really quite good. For example, the projected demand for AC5010 is zero for both the Winter and Spring quarters, and the course is not offered in either of these periods. ST5020 has only a projected demand of 0 and 3 students in the Fall and Summer quarters

respectively; and the actual schedule has the course only offered in the Winter and Spring, where the projected demands are 100 and 117. A similar analysis shows good course scheduling for all courses except BL5010, MG5020, and MK5020. In BL5010, the projected demand for Winter quarter is 56 students. However, the course is not offered in the Winter but rather in the Summer where the demand is only 16. Similarly, MG5020 has a projected demand in the Winter quarter of 45 students, but the course is offered in the Spring where the projected demand is only 7. And finally, MK5020 is not offered in the Winter when the demand is 55 but rather in the Fall where the demand is only 10.

Obviously, there are other factors that were considered when the actual course schedule was generated. However, these discrepancies should be studied to see if course changes are advisable.

The simulation was also run in evaluation mode using the course schedule from Table 5.3 with infinite class capacities. The results are given in Table 5.5.

This table is used primarily to predict course enrollments for staffing considerations. It does suggest a few course offerings that might be deleted. For example, MK5020 in the Fall quarter only has a projected demand of 5 students and MS5010 in the Summer quarter only a demand of 19.

This schedule does present some problems for students taking 3 courses per quarter with a nonbusiness undergraduate degree. A typical schedule for a student entering in the Fall is given in Table 5.6.

Table 5.5. Enrollment Figures for the Simulation in Evaluation Mode

	Quarter			
Course	Fall	Winter	Spring	Summer
AC5010	116	0	0	39
AC5020	153	47	0	0
AC5030	0	134	49	0
BL5010	60	0	0	53
EC5010	116	0	0	39
EC5020	145	114	0	0
EC5030	0	0	117	59
FN5010	0	135	33	0
FN5020	0	0	78	73
MG5010	52	41	0	101
MG5020	57	0	55	61
MS5010	0	71	123	19
MK5010	68	63	64	0
MK5020	5	0	69	90
ST5010	135	126	0	0
ST5020	0	99	117	0
BA5310	59	27	0	57

Table 5.6. Courses Taken in Each Quarter for a Student with a Nonbusiness Undergraduate Degree Entering in the Fall Quarter

Quarter	Courses Taken
Fall	AC5010, BL5010, EC5010
Winter	MG5010, AC5020, ST5010
Spring	ST5020, MS5010, AC5030
Summer	EC5030, MG5020
Fall	EC5020, MK5010
Winter	FN5010
Spring	FN5020, MK5020
Summer	BA5310

As is suggested in Table 5.6, it takes 8 quarters to complete the core courses. Indeed, a student in this situation would start taking concentration and elective courses earlier to fill up his schedule so it may actually cause no delay—but the possibility for delay exists. The reason for the rather empty schedule is that the course schedules were designed for students taking four courses per quarter rather than three. This, however, is the only student classification that incurs significant delays in completing the core courses under the actual schedule.

Even after it has been decided which courses to offer in a given quarter, the problem of timetabling, or assigning courses to particular times, still exists. To aid in this task, the simulation model provides conflict matrices for each quarter. These are given in Figures 5.2 through 5.5.

If it is the case that two courses must be offered at the same time, these matrices can be used to minimize student conflicts.

Hopefully, in the near future more accurate data will be available; and the simulation model can be rerun, thus providing the administrators with useful tools in scheduling courses.

	BA5310	ST5510	MK5020	MK5010	MG5020	MG5010	EC5020	EC5010	BL5010	AC5020
AC5010	0	0	0	0	0	17	0	57	38	0
AC5020	0	38	0	37	28	17	43	0	0	
BL5010	0	3	0	0	0	17	0	38		
EC5010	0	0	0	0	0	17	0			
EC5020	0	53	0	22	0	12				
MG5010	0	17	0	0	0					
MG5020	0	0	0	0						
MK5010	0	8	0							
MK5020	0	0								
ST5010	0									

Figure 5.2. Conflict Matrix for Fall Quarter.

	BA5310	ST5210	ST5010	MK5010	MS5010	MG5010	FN5010	EC5020	AA5030
AC5020	0	0	45	17	0	24	0	24	0
AC5030	0	20	0	22	8	13	90	0	
EC5020	0	5	54	22	0	3	0		
FN5010	0	20	0	12	8	13			
MG5010	0	0	24	0	0				
MS5010	0	32	0	0					
MK5010	0	17	17						
ST5010	0	0							
ST5020	0								

Figure 5.3. Conflict Matrix for Winter Quarter.

	AC5030	EC5030	FN5020	FN5010	MG5020	MS5010	MK5020	MK5010
ST5020	41	13	20	0	0	68	16	0
MK5020	0	20	0	35	28	12	0	
MK5010	7	33	7	0	0	0		
MS5010	43	12	22	12	0			
MG5020	0	28	0	9				
FN5020	0	20	0					
FN5010	29	0						
EC5030	0							

Figure 5.4. Conflict Matrix for Spring Quarter.

	AC5010	BL5010	EC5030	EC5010	FN5020	MG5020	MG5010	MS5010	MK5020
BA5310	0	0	0	0	0	0	0	0	0
MK5020	0	0	0	20	53	20	17	16	
MS5010	0	0	0	3	16	0	0		
MG5020	0	0	0	43	22	0			
MG5010	5	24	5	15	0				
FN5020	0	0	0	25					
EC5030	0	0	0						
EC5010	19	12							
BL5010	12								

Figure 5.5. Conflict Matrix for Summer Quarter.

CHAPTER 6

OVERVIEW OF PERIODIC SCHEDULING

A. INTRODUCTION

In many different types of scheduling problems, it is useful to restrict the type of schedule considered when searching for an optimal solution. For example, consider the flow shop scheduling problem. In this problem, there are m different machines and n jobs each consisting of m different tasks which must be performed on different machines. On each machine, there are n! possible sequences to be considered. Since there are m machines, the total number of possible schedules is $(n!)^m$. Consequently, if some of these possibilities could be eliminated, it would greatly simplify the problem.

For the special case when there are only 2 machines, permutation schedules form a dominant set for any regular measure. A permutation schedule is a schedule in which the same job sequence occurs on all machines. Consequently, a complete schedule can be characterized by the permutation of n integers. This result was used by Johnson [15] to develop an 0 (nlogn) time algorithm for the two machine flow shop problem where F_{max} is to be minimized.

If $m \geq 3$, then permutation schedules no longer form a dominant set. In fact, Garey, Johnson, and Sethi [10] proved the problem to be NP-complete. However, for the ease of characterizing and implementing a schedule, it may still be useful to restrict the search for a solution to

a permutation schedule. Even with this restriction, the permutation flow shop problem is NP-complete. Nevertheless, Lageweg, Lenstra, and Rinnooy Kan [16] developed a branch and bound scheme that works well as long as the number of machines is moderately small. The success of the algorithm is due to the tight bounds that can be calculated when only permutation schedules are considered. If nonpermutation schedules are allowed, the only bounds that can be calculated are quite loose causing the branch and bound scheme to enumerate an enormous number of schedules.

The point of the example is that although the restriction to permutation schedules may result in a nonoptimal solution to the general flow shop problem, there are still valid reasons for this restriction. A permutation schedule is easy to characterize and implement, whereas a nonpermutation schedule may not be. Also, a moderately efficient algorothm exists for the permutation flow shop where none exists for the general flow shop short of almost complete enumeration.

B. PERIODIC SCHEDULES

In the case of academic course scheduling, another type of schedule restriction might be useful. Consider a course that is only offered once per year as opposed to every quarter. It would be possible to schedule the course in the fall quarter one year, in the winter quarter another year, and so on. This type of schedule would cause considerable confusion for students trying to determine their long-term course schedules and also for administrators trying to staff the various courses. Consequently, in most academic institutions, if a course is

only offered once per year, it is offered in the same quarter each year. It may be the case that for some courses, only one offering every other year would be sufficient. Nevertheless, that course will usually be offered in the same quarter every other year. Schedules of this type have a periodic property because there is a period during which the course offerings repeat. To be more precise, let S_k be the set of courses offered in period K, $K \in \{0,1,2,\ldots\}$. S_k is a <u>periodic schedule</u> if there exists a positive integer n such that

$$S_k = S_{[k]_n}$$

where $[k]_n$ is k modulo n.

There are many situations besides academic course scheduling for which periodic schedules are appropriate. The primary motivation is usually to simplify the schedule. However, it may be the case that the students or jobs arrive in a periodic fashion. If N_k is the number of students or jobs arriving in period K, then the arrival function is said to be periodic if

$$N_k = N_{[k]_n}$$

for some n. For student arrivals, the period length is one year. Indeed, there may be some growth trends in student arrivals, but the underlying, periodic nature of the arrivals still remains. Consequently, besides the desire to simplify the nature of the problem by considering

only periodic schedules, periodic arrivals tend to induce the periodic schedule.

C. GENERAL ACADEMIC SCHEDULING PROBLEM

The general academic scheduling problem can be modeled as a periodic scheduling problem. Let C denote the set of courses $\{C_1, C_2, C_3, \ldots, C_q\}$. Also, there exists a partial ordering $<$, called the prerequisite structure, such that if $C_i < C_j$, then C_i must be completed before C_j can be started. In general, the prerequisite structure has no special structure that can be exploited. Also, associated with each course i is a maximum enrollment e_i.

Each student follows a program of study P(i) consisting of a subset of C. For example, if $P(2) = \{C_1, C_3, C_4\}$, then the second program of study, or curriculum, consists of three courses: C_1, C_3, and C_4. Let $P = \{P(1), P(2), \ldots P(r)\}$ denote the set of all possible curricula.

In each time period, the number of students who enter under each curriculum may vary. Let N_{it} denote the number of students entering in period t planning to follow the i^{th} program. It is useful to restrict the arrival function to periodic arrivals. Therefore

$$N_{it} = N_{i[t]_n}$$

for some n. Now if the class capacities e_i are large compared to the number of students entering, further simplification results. It may be

the case that the students entering in the same period under the same curriculum may graduate in different periods. This would be caused by one of the students encountering a full class, and consequently having to delay graduation. Consequently, if the capacities are small compared to the demand for class, each student would have to be considered individually. This situation was handled in earlier chapters by using the simulation model. However, if the capacities are assumed to be large, all students entering in the same period under the same curriculum will graduate at the same time. The arrival function can then be considered a weight function proportional to the number of students arriving each period. If an equal number of students arrive each period, weights of one could be assigned to all periods, greatly simplifying the problem. The weight function can also be viewed as the relative importance of students entering in different periods. Let w_{it} be the weight attached to a student entering in period t following curriculum i. Again, this function is assumed to be periodic.

Let f_{it} be the time required for a student starting in period t following curriculum i to graduate. The problem then is to determine a schedule $S = \{S_1, S_2, \ldots\}$ where S_t is the set of courses offered in period t such that

$$\sum_{i=1}^{r} \sum_{t=1}^{\infty} w_{it} f_{it}$$

is minimized. If only periodic schedules are considered, then only n

sets of courses $\{S_1, S_2, \ldots, S_n\}$ are required to completely characterize the schedule since

$$S_k = S_{[k]_n}$$

Consequently, the objective can be simplified to

$$\text{Min} \sum_{i=1}^{r} \sum_{t=1}^{n} w_{it} f_{it}$$

There are additional constraints for the problem. Usually, an academic institution has limited resources, and only a certain number of courses can be offered each period. Let M_t be the maximum number of courses that can be offered in period t. Then, for a schedule to be feasible,

$$|S_t| \leq M_t$$

where $|\cdot|$ denotes cardinality.

In summary, the problem can be stated as:

$$\text{Min} \sum_{i=1}^{r} \sum_{t=1}^{n} w_{it} f_{it}$$

$$\text{S. T.} \quad |S_t| \leq M_t \qquad \text{for } t = 1,2,\ldots n$$

$$S_t \subset C \qquad \text{for } t = 1,2,\ldots n$$

Notice that it is not clear how to calculate f_{it} nor should it be, since the problem facing the student of deciding which courses to take in which periods is in general NP-complete.

In particular, suppose a student can take a maximum of m courses per period. Also, assume that each course is offered every period. Then the problem facing the student is the m/arbitrary/1/F_{max} scheduling problem which, for arbitrary m, is NP-complete. The student's problem has the added complication that not all courses are offered every period.

CHAPTER 7

THE CLOCK PROBLEM

A. INTRODUCTION

In order to develop some theoretical groundwork for analyzing periodic schedules, it is useful to consider the following simple problem denoted by the Clock Problem (CP). In this problem, all jobs are identical and consist of only one task requiring unit execution time. Also, in any given set of n time periods, only m processors are available.

For example, suppose that only three processors can be used in every eight time periods. One possible solution would be to schedule the processors in periods 0, 2, and 4 as shown in Figure 7.1.

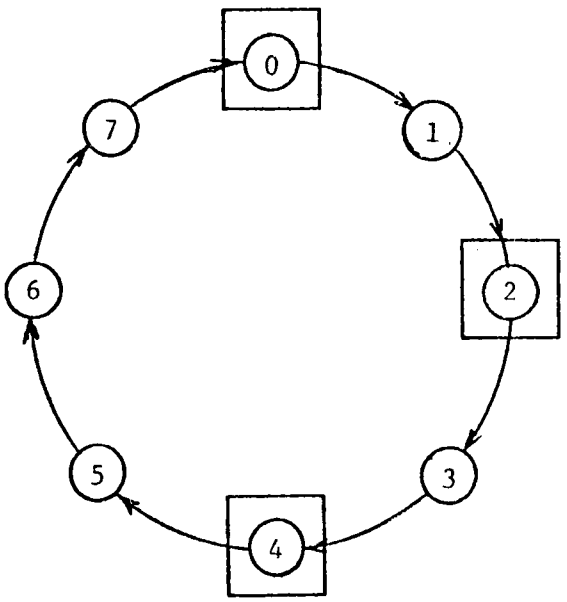

Figure 7.1. Example Clock Problem.

Each processor is assumed to have infinite capacity; and since there is only one task, there is no problem concerning the processors being set up for the wrong task. Consequently, a job arriving in period 0 will be processed immediately. However, a job arriving in period 1 will have to wait until period 2 to be processes. Similarly, a job arriving in period 5 will have to wait until period 0 to be processed. (Remember that because of the periodic nature, period 0 immediately follows period n-1.)

Since the job requires only one period to be processed, any job that requires more than one period is said to incur delay. For the clock problem, the delay d_j of a job arriving in period j is the number of time periods until its execution begins. The delays in the example are given by $(d_0, d_1, \ldots d_7) = 0, 1, 0, 1, 0, 3, 2, 1)$.

The objective of the clock problem is to minimize total weighted delay D where

$$D = \sum_{j=0}^{n-1} w_j d_j$$

In the situation where only one job (or equivalently an equal number of jobs) arrive in each period, the weights w_j can be assumed to be 1. This problem will be known as the unweighted clock problem or just the clock problem (CP). In the example, the total delay for the CP is 8. Section B will consider the unweighted clock problem, while in Section C, the weighted clock problem (WCP) will be discussed.

B. THE UNWEIGHTED CLOCK PROBLEM

The CP can be formulated as an integer linear program. Since there are m processors to be scheduled, there must be m gaps (possibly of length zero) between processors. If d_k is the number of periods in the kth gap, the total delay caused by a gap of length d_k is $\sum_{t=0}^{d_k} t$.

Then the CP can be written as

$$\text{minimize} \quad \sum_{k=1}^{m} \sum_{t=1}^{d_k} t \tag{7.1}$$

$$\text{s.t.} \quad \sum_{k=1}^{m} d_k = b \tag{7.2}$$

$$d_k \geq 0, \text{ integer} \tag{7.3}$$

where $b = n-m$.

Now, since $\sum_{t=1}^{d_k} t = \frac{d_k(d_k + 1)}{2}$, (7.1) can be written as $\sum_{k=1}^{m} \frac{d_k(d_k + 1)}{2}$, which from (7.2) is $\frac{1}{2} \sum_{k=1}^{m} d_k^2 + \frac{b}{2}$.

Let $\left[\frac{b}{m}\right]$ denote the integer part of $\frac{b}{m}$ and let $x_k = d_k - \left[\frac{b}{m}\right]$. From (7.2), the CP can be relaxed as

$$\text{minimize} \quad \sum_{k=1}^{m} x_k^2 + Q$$

$$\text{s.t.} \quad \sum_{k=1}^{m} x_k = b - m\left\lfloor \frac{b}{m} \right\rfloor \tag{7.4}$$

$$x_k \text{ integer}$$

where Q is the constant $2b\left\lfloor \frac{b}{m} \right\rfloor - m\left\lfloor \frac{b}{m} \right\rfloor^2 + b$.

The model (7.4) is equivalent to (7.1)-(7.3) except for the relaxation of the requirement of $d_k \geq 0$.

The solution to (7.4) is obvious and is to let exactly $b - m\left\lfloor \frac{b}{m} \right\rfloor$ x's equal 1 and all other x's equal 0. Consequently, if $x_k = 1$, then $d_k = \left\lfloor \frac{b}{m} \right\rfloor + 1$ and if $x_k = 0$, then $d_k = \left\lfloor \frac{b}{m} \right\rfloor$.

Since this solution satisfies $d_k \geq 0$, it is an optimal solution to CP. The property that an optimal solution to CP contains gaps of at most two different lengths will be used in Chapters 8 and 9.

C. WEIGHTED CLOCK PROBLEM

Unfortunately, for the weighted clock problem, very few results have been obtained. One result is that if there are weights of zero associated with certain periods, then processors should never be located in those periods.

Theorem 7.1: For the WCP, the set of periods in which $w_j > 0$ form a dominant set.

Proof: Let $S = \{j | w_j = 0\}$ and $S' = \{j | w_j > 0\}$

Suppose that a processor is located at $k \epsilon S$. Further, let $P^k = \{j |$ all jobs arriving in period j are executed on the processor located in period k$\}$. Consider the example in Figure 7.2 where the squares indicate processor locations.

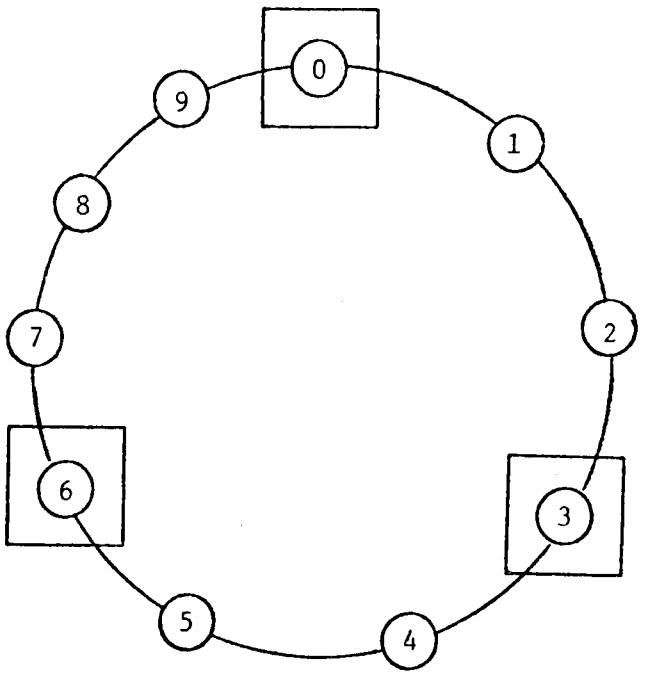

Figure 7.2. Example Weighted Clock Problem.

Jobs arriving in periods 1, 2, or 3 are executed by the processor located in period 3. Consequently, $P^3 = \{1,2,3\}$. Similarly, $P^6 = \{4,5,6\}$ and $P^0 = \{7,8,9,0\}$.

The total delay for all jobs in P^k is given by

$$D^k = \sum_{j \epsilon P^k} w_j [k-j]_n$$

where $[k-j]_n$ is (k-j) modulo n.

If, in the example, $(w_0, w_1, \ldots w_9) = (1,1,1/2,1,1/2,1,0,1,1,1/2)$, then $D^3 = 3$, $D^6 = 2\text{-}1/2$, and $D^0 = 5\text{-}1/2$.

Now suppose that the processor located in period k is moved to period k' which is the first period encountered in the counter clockwise direction such that $k' \epsilon S'$. The new total delay for jobs in $P^{k'}$ is

$$D^{k'} = D^k - \sum_{j \epsilon P^k} [k-k']_n$$

Since $w_j \geq 0$ and $[k-k']_n > 0$, the term on the right is always nonnegative. Since the delay of all other jobs executed on other processors is not affected by the shift, the shift will never cause an increase in total delay. Consequently, S' forms a dominant set.

In the example, if the processor located in period 6 is shifted to period 5, then $D^5 = 1$. Notice that D^3 and D^0 remain unchanged causing the total delay to decrease by 1-1/2 units.

The WCP with m = 1 has an analogy in location theory. There is a well-known problem in location theory known as the "ice cream man problem." This problem is to locate a service facility (an ice cream man) on a line (beach) along which there are various demand points (sunbathers). To each demand point is attached a weight which is proportional to the number of people or importance of the people at that

point. The problem is to locate the facility such that the weighted distance to all demand points is minimized. The WCP can be viewed as an "ice cream man on a circle" problem. In the linear ice cream man problem, it can be shown that the facility should always be located at a demand point. The analogous result was demonstrated for the WCP in Theorem 7.1.

The optimal solution for the ice cream man problem is easy to find using the median location which is defined to be the location such that no more than one-half of the weights are to the right and no more than one-half are to the left. The following example illustrates the median rule. In Figure 7.3, the nodes represent demand points; the weights are given above each node, and the locations are given below each node.

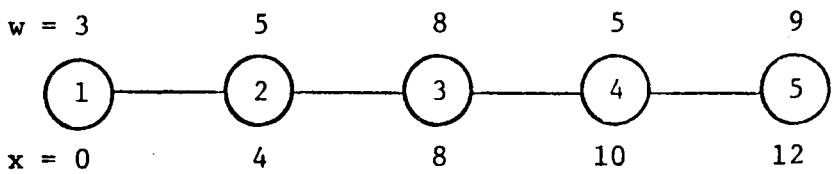

Figure 7.3. Example Ice Cream Man Problem.

The sum of the weights is 30. Thus, the median location corresponds to a cumulative of 30/2 = 15. Table 7.1 shows the cumulative weights for each node.

At node 3, the cumulative weights first exceed the median value of 15, and consequently, is the optimal location. This solution technique works because of the existence of only one local, and consequently, global optimum. The weighted total distances are given in Table 7.2, where node 3 is the optimum.

Table 7.1. Ice Cream Man Solution

Node	Location	Weight	Cumulative Weight
1	0	3	3
2	4	5	8
3	8	8	16
4	10	5	21
5	12	9	30

Table 7.2. Weighted Distances for Locations at Each Node

Node	Total Weighted Distances
1	242
2	146
3	90
4	94
5	118

On the other hand, the weighted clock problem may have several local optima as shown in the following example. In Figure 7.4, the numbers beside each period j represent the weights w_j.

The total delays for various possible solutions are given in Table 7.3.

In this example, there are three local optima at periods 0, 2, and 6. Even if the weights are restricted to either 0 or 1, local optima may still exist. This is demonstrated in Figure 7.5.

Table 7.4 gives the total weighted delays for various solutions. From the results of Theorem 7.1, only those periods with weights equal to one need be considered as possible solutions. For this example, local optima exist in periods 1, 7, and 12.

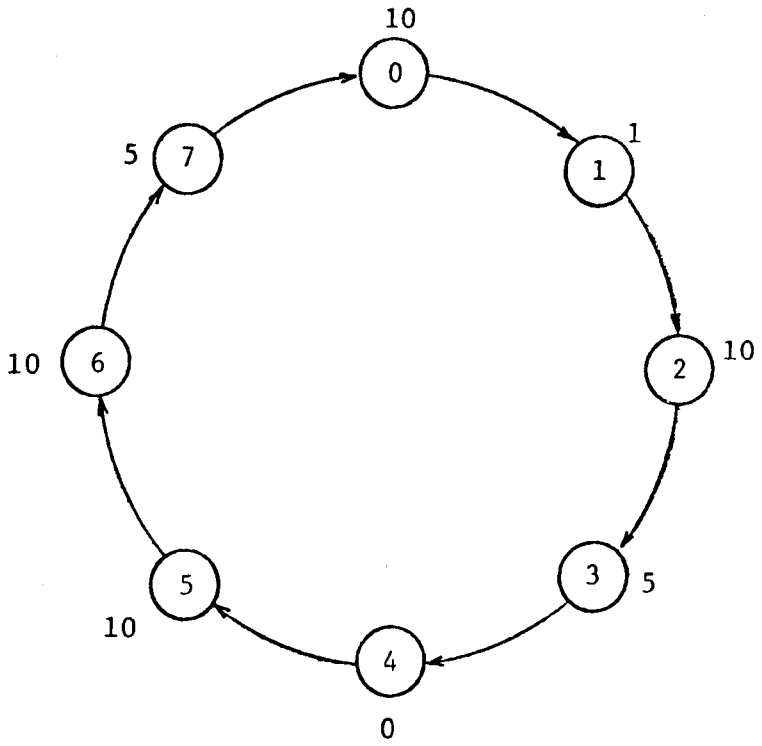

Figure 7.4. Example Weighted Clock Problem with Local Optima.

Table 7.3. Example of Local Optima for Weighted Clock Problem

Solution Period	Total Weighted Delay
0	147*
1	195
2	161*
3	172
4	223
5	194
6	165*
7	176

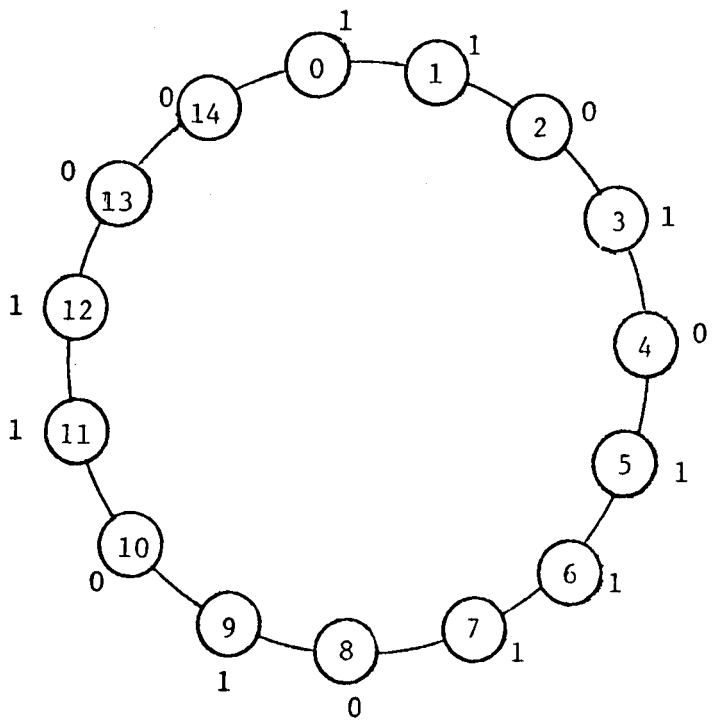

Figure 7.5. Example Weighted Clock Problem with Zero-One Weights.

Table 7.4. Example of Local Optima for Weighted Clock Problem with 0-1 Weights

Solution Period	Total Weighted Delay
0	66
1	60*
3	63
5	66
6	60
7	54*
9	57
11	60
12	54*

One result that is possible when the weights are restricted to 0 or 1 is stated in the following theorem.

Theorem 7.2: In a string of consecutive periods with weights equal to one, only the most clockwise period in the string need be considered as a solution candidate.

Proof: Consider two consecutive periods p_1 and p_2 with weights equal to one where p_2 is the more clockwise period. Let $D(p_1)$ = the total weighted delay if the processor is located in period p_1. If the processor is shifted one period in the clockwise direction to p_2, then the shift will cause an increase in delay of one period for all periods j with $w_j = 1$ with the exception of p_2. For p_2, the delay will decrease from n-1 to 0. Consequently,

$$D(p_2) = D(p_1) + (W-1) - (n-1)$$

$$= D(p_1) = W-n$$

where W is the number of periods with $w_j = 1$, and n is the number of periods. Now, since $n > W$, except for the unweighted case, where all periods have weights equal to one, then $D(p_2) < D(p_1)$. Therefore, the clockwise shift will always decrease the total weighted delays, and the theorem is established.

This is illustrated in the previous example in the strings (0,1), (5,6,7), and (11,12) where each shift to a more clockwise period causes a change in total weighted delay of W-n = 9-15 = -6.

The only result that has been obtained for the WCP is to restrict the possible solution periods to a smaller set. For the $m = 1$ case, these results are sufficient to make the problem relatively easy to solve. However, for the $m > 1$ case, a rather large combinatorial problem results with the only solution technique found so far being an enumerative scheme.

CHAPTER 8

SERIAL PERIODIC SCHEDULING PROBLEM

A. PROBLEM STATEMENT

In the Serial Periodic Scheduling Problem (SPSP), all jobs are identical and consist of ℓ tasks with a series precedence structure. That is, task i + 1 cannot be performed unless task i was performed in an earlier period. Figure 8.1 depicts a series precedence structure.

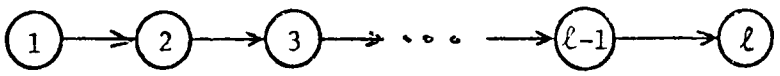

Figure 8.1. Series Precedence Structure.

Each processor can execute only one type of task at a time. Consequently, it is necessary to introduce another parameter for the SPSP. Let p be the maximum number of processors that are available each period. Since each processor is assumed to have infinite capacity for the task it is set up to process, only the situation where $p < \ell$ is of interest. As in the clock problem, total delay is to be minimized. However, only the unweighted SPSP will be considered.

In the following example, n = 8, p = 2 and ℓ = 5. If the tasks are denoted A, B, C, D, and E, a nonoptical solution is shown in Figure 8.2.

The letters beside each time period represent processors capable of executing that type of task. The delays in this example are given by $(d_1, d_2, \ldots d_7) = (0,5,4,3,2,1,0,1)$.

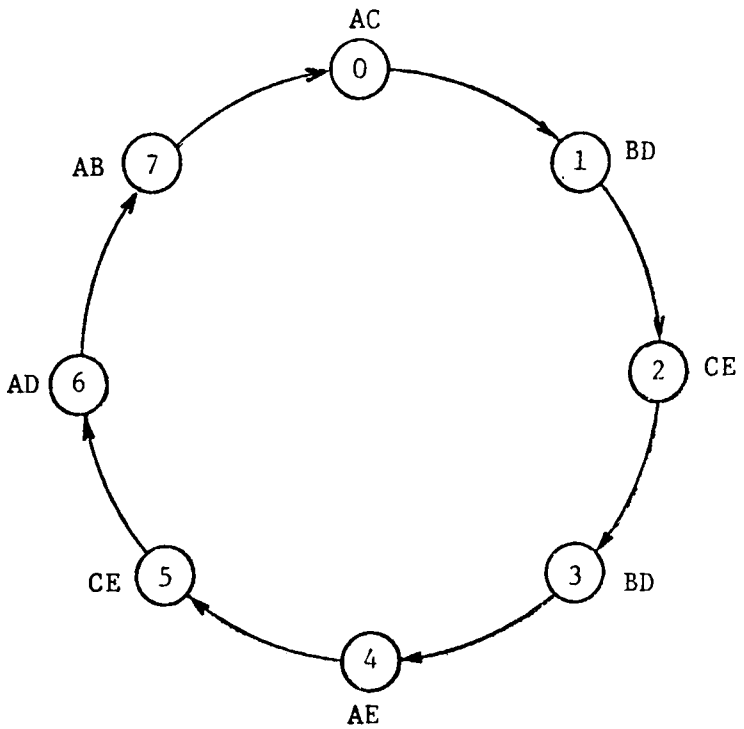

Figure 8.2. SPSP Example Solution.

B. PRIME AND CONTIGUOUS SCHEDULES

Define a prime schedule to be a schedule such that the removal of any task from the schedule causes an increase in D.

Theorem 8.1: In a prime schedule for the SPSP, each task is performed the same number of times.

Proof: Let the set of time periods in which the first task is performed be T_1, and denote the other periods by $\overline{T}_1$. It will be shown that the schedule contains exactly $|T_1|$ sequences, each containing all ℓ tasks, where $|\cdot|$ denotes cardinality.

Consider task k which is performed in time period i_k. Associated with k there must be a unique element of T_1, say $T_1(k)$, which is the latest time in which a job can start and still have task k performed in period i_k. Uniqueness of $T_1(k)$ follows immediately from the requirement that the schedule be prime, otherwise some elements of T_1 would contain an extraneous task.

Similarly, there is a unique earliest finishing time i_t associated with each element t of T_1. A unique sequence of ℓ tasks achieves i_t, for if more than one sequence did so, then some tasks would be extraneous.

Thus it has been shown that a unique starting time is associated with each task and a unique sequence for each starting time. Consequently, the theorem is established.

Given that each task is performed $|T_1|$ times, $|T_1|$ may be made as large as possible. Clearly an upper bound on $|T_1|$ is $\frac{np}{\ell}$, and we take $|T_1|$ to be $\left\lfloor \frac{np}{\ell} \right\rfloor$ and denote that quantity by c.

A sequence of length ℓ is contiguous if task i performed in period j implies that task i + 1 is performed in period $[j + 1]_n$ for $i = 1,2, \ldots , \ell-1$. Only contiguous sequences will be considered, which allows a schedule to be completely characterized by the period in which each sequence begins. It will be shown later that this restriction causes no difficulty since there is always an optimal schedule consisting only of contiguous sequences.

C. AN OPTIMAL SOLUTION TO SPSP

In this section, the existence of a feasible contiguous schedule will be shown, the c starting periods of which solve the associated CP. Such a solution is clearly optimal to SPSP, since its only delay occurs in waiting for the starting point of each sequence and since a solution to the CP yields a lower bound for this delay.

Before demonstrating this optimal solution, notice that it does not suffice to find any optimal solution to the CP and then begin the c sequences at these points. For example, if n = 10, ℓ = 5, p = 2, then c = 4. An optimal solution to CP is {0,2,4,7} which is infeasible to SPSP since three jobs are scheduled in period 4. This is shown in Figure 8.3.

0	1	2	3	[4	5	6	7	8	9
A	B	C	D	E					
		A	B	C	D	E			
				A]	B	C	D	E	
D	E						A	B	C

Figure 8.3. An Infeasible Solution to SPSP.

Theorem 8.2: The schedule $S = \left\{0, \left[\frac{n}{c}\right], \left[\frac{2n}{c}\right], \ldots, \left[\frac{(c-1)n}{c}\right]\right\}$ is optimal to SPSP.

Proof: The proof is in two parts. It is necessary to first show that S is feasible and then that it solves the associated CP.

S is feasible

Note that $p < \ell <$ np because if $\ell <$ np, there is no feasible schedule. Since the schedule is periodic, the first task in each series will be executed at times $\left\lfloor \frac{kn}{c} \right\rfloor$ for k = 0,1,2, ... , (c-1). It suffices to show that for any value of $k \geq 0$ that the k^{th} and the $(k + p)^{th}$ occurrence of the first task are separated by a length of time no less than ℓ. This is demonstrated as follows:

$$\left\lfloor \frac{(k + p)n}{c} \right\rfloor - \left\lfloor \frac{kn}{c} \right\rfloor \geq \left\lfloor \frac{kn}{c} \right\rfloor + \left\lfloor \frac{pn}{c} \right\rfloor - \left\lfloor \frac{kn}{c} \right\rfloor = \left\lfloor \frac{pn}{\left[\frac{np}{\ell}\right]} \right\rfloor \geq \left\lfloor \frac{pn}{\left[\frac{np}{\ell}\right]} \right\rfloor = \ell$$

where the first inequality follows from the general result $[x + y] \geq [x] + [y]$.

S solves CP

An arbitrary gap between consecutive elements of S is given by

$$g = \left\lfloor \frac{(i + 1)n}{c} \right\rfloor - \left\lfloor \frac{in}{c} \right\rfloor - 1$$

where $i \in \{0,1, \ldots , c-1\}$.

If $\frac{in}{c}$ is integer, then $g = \left\lfloor \frac{n}{c} \right\rfloor - 1$. If $\frac{in}{c}$ is noninteger, then either $\left\lfloor \frac{(i + 1)n}{c} \right\rfloor = \left\lfloor \frac{in}{c} \right\rfloor + \left\lfloor \frac{n}{c} \right\rfloor$ so that $g = \left\lfloor \frac{n}{c} \right\rfloor - 1$ or $\left\lfloor \frac{(i + 1)n}{c} \right\rfloor = \left\lfloor \frac{in}{c} \right\rfloor + \left\lfloor \frac{n}{c} \right\rfloor + 1$ and $g = \left\lfloor \frac{n}{c} \right\rfloor$.

Since gaps are only two lengths which differ by one, S solves CP and the theorem is proved.

CHAPTER 9

EMPTY PRECEDENCE PERIODIC SCHEDULING

A. PROBLEM STATEMENT

It may be the case that the tasks in a job are order independent. In other words, there is no precedence structure between the tasks. This problem is called the Empty Precedence Periodic Scheduling Problem (EPPS).

In the previous chapter, the serial precedence structure implied that each job could have only one task being executed at a time. However, in the empty precedence case, several tasks in a job may be executed simultaneously. Consequently, an additional parameter is needed to set an upper limit on the number of tasks that can be executed simultaneously for a particular job. In the academic course scheduling problem, this parameter refers to the maximum number of courses a student can take in a quarter.

The Empty Precedence Periodic Scheduling Problem can be stated as follows: In each of the n periods, there are a maximum of m processors available. There are ℓ independent tasks in each job, and each job can have at most p tasks executing simultaneously in any period. Each processor should be assigned to execute a particular type of task such that the total delay for all jobs is minimized.

As in the serial case, only the unweighted problem will be considered. In most scheduling situations, there is no reason to place

a limit on the number of tasks of a job that can be executing simultaneously. However, there is a limitation implied by the number of processors. Consequently, if $p \geq m$, then only the $p = m$ case need be considered. An optimal algorithm for the $p = m$ case is presented in this chapter. Unfortunately for the $m > p$ case, no algorithm has yet been found; and further research is needed on this subproblem. In the remainder of this chapter, the Empty Precedence problem will refer to the $m = p$ case.

B. SOME USEFUL PROPERTIES FOR THE EMPTY PRECEDENCE CASE

In searching for an optimal algorithm for the empty precedence case, several characteristics need to be considered. Processors can be set up to execute a specific task from 0 to np times since there are a total of np processor assignments to be made. However, all but a few policies are clearly suboptimal. After a processor is set up or scheduled for a particular task, it is assumed to be able to execute an infinite number of those tasks, therefore there is never any need to have two processors set up to execute the same task in a given period. If there are ℓ tasks in a job and only p processors, then the minimum number of periods required to complete all the tasks is $\left\langle \frac{\ell}{p} \right\rangle$ where $\langle x \rangle$ denotes the smallest integer $\geq x$. A task that cannot be executed in any set of $\left\langle \frac{\ell}{p} \right\rangle$ consecutive periods will cause delay and is called a delay task. In Figure 9.1, course D is a delay task because it cannot be processed in either period 0 or 3.

Period			
0	1	2	3
A	D	B	E
B	E	C	A
C	A	D	B

Figure 9.1. Example of a Delay Task.

Consequently, a job arriving in period 3 must wait until period 1 for task D to be executed. This job has a flowtime of 3, whereas the minimum flowtime is $\left\langle \frac{5}{3} \right\rangle = 2$. Any job arriving in period 3 then would have a delay of 1.

It is obvious that a task is a delay task if it is processed infrequently or if the times it can be processed are not evenly spaced. To ease the explanation of schedules, a task is said to be scheduled in a period if a processor is set up to execute that type of task in that period. Since there are ℓ tasks and np processors to be scheduled, each task should be scheduled either $\left\lfloor \frac{np}{\ell} \right\rfloor$ or $\left\lfloor \frac{np}{\ell} \right\rfloor + 1$ times. If a task is scheduled $\left\lfloor \frac{np}{\ell} \right\rfloor + 1$ times, it is possible for that task to never be a delay task because it can be scheduled in each set of $\left\langle \frac{\ell}{p} \right\rangle$ consecutive time periods as shown below.

$$\left\langle \frac{\ell}{p} \right\rangle \left(\left\lfloor \frac{np}{\ell} \right\rfloor + 1 \right) > \left\langle \frac{\ell}{p} \right\rangle \left(\frac{np}{\ell} \right) \geq \left(\frac{\ell}{p} \right) \left(\frac{np}{\ell} \right) = n$$

The tasks that are scheduled $\left\lfloor \frac{np}{\ell} \right\rfloor$ times, however, may or may not be delay tasks. In Figure 9.2, tasks C, D, E, and F are scheduled $\left\lfloor \frac{5.4}{6} \right\rfloor = 3$ times, and yet none are delay tasks.

Period				
0	1	2	3	4
A	E	C	A	E
B	F	D	B	F
C	A	E	C	A
D	B	F	D	B

Figure 9.2. Example of Nondelay Tasks.

However, to schedule a task less than $\left\lceil \frac{np}{\ell} \right\rceil$ times assures that this task will be a delay task.

$$\left(\left\lceil \frac{np}{\ell} \right\rceil - 1 \right) \frac{\ell}{p} < \left\lceil \frac{np}{\ell} \right\rceil \left(\frac{\ell}{p} \right) \leq \left(\frac{np}{\ell} \right) \left(\frac{\ell}{p} \right) = n$$

In the serial periodic scheduling problem, it was shown that an optimal schedule could always be found in which each task was scheduled the same number of times. In the empty precedence problem, a slightly more general property holds. If there are just two tasks to be scheduled, an optimal policy for scheduling them in an infinite number of periods is to alternate the schedulings between the two tasks in the available period. This is obvious because a job arriving in any period will encounter processors set up to execute both tasks before a processor set up to execute a repetitious task is encountered. For example, in Figure 9.3, a job arriving in period 2 would encounter processors set up to execute task B twice before task A could be processed.

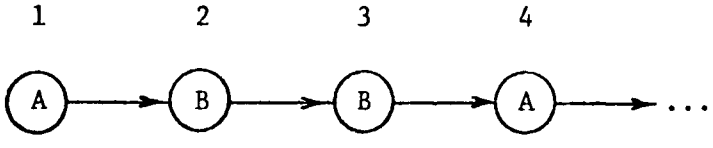

Figure 9.3. A Suboptimal Schedule.

An optimal policy would be to alternate the task schedules as shown in Figure 9.4.

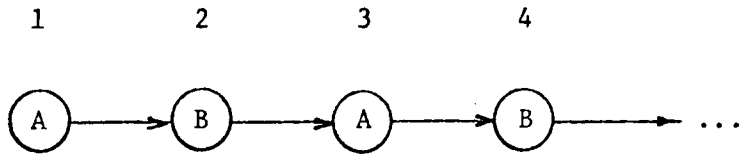

Figure 9.4. An Optimal Policy.

When the periodic nature of the problem is added, however, it is not always possible to alternate the task schedules. In scheduling two tasks, if the number of periods in which the tasks are to be scheduled is odd, then there will exist two consecutive periods in which the same task is scheduled. In Figure 9.5 below, jobs arriving in period 4 encounter a delay because task A is scheduled in two consecutive periods.

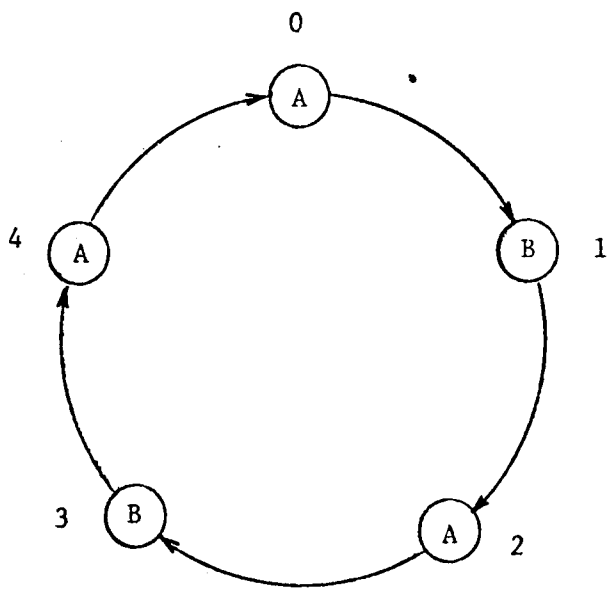

Figure 9.5. Delay Caused by an Odd Number of Periods.

In fact, whenever the same task is scheduled twice between the scheduling of another task, delay is encountered. The process of alternating the task schedules motivates the following theorem.

Theorem 9.1: An optimal schedule for the empty precedence periodic scheduling problem can always be found in which each task is scheduled either k or k + 1 times.

Proof: Consider two tasks i and j such that task i is scheduled n_i times and task j is scheduled n_j times. Assume an optimal schedule in which $|n_i - n_j| > 1$. Without loss of generality, let $n_i > n_j + 1$. Then, either of two possible task structures must exist.

1. Task i is scheduled 3 times or more without task j being scheduled. This is depicted in Figure 9.6.

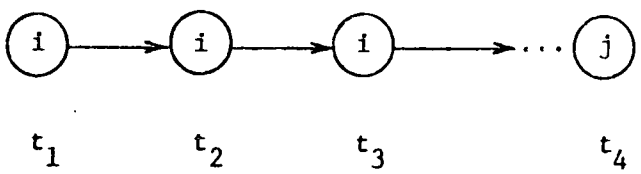

Figure 9.6. Three Consecutive Processors for the Same Task.

2. Task i is scheduled at least 2 times without task j being scheduled on at least two different occasions. This is shown in Figure 9.7.

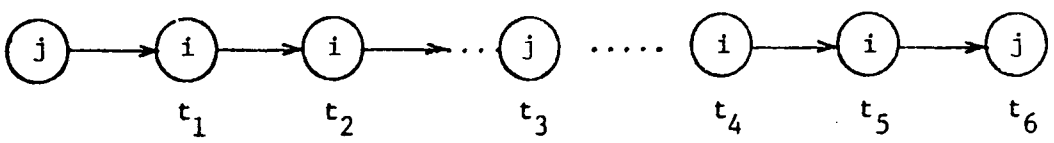

Figure 9.7. Multiple Occurrences of Two Consecutive Processors for the Same Task.

Case 1: The time to complete tasks i and j in a job arriving in t_1, t_2, or t_3 will be $[t_4 - t_1]_n$, $[t_4 - t_2]_n$, and $[t_4 - t_3]_n$ respectively. If the task scheduled at t_2 is changed to task j, these flowtimes become $[t_2 - t_1]_n$, $[t_3 - t_1]_n$, and $[t_4 - t_2]_n$. Now since $[t_4 - t_2]_n = [t_3 - t_2]_n + [t_4 - t_3]_n$, the change improves the flowtimes to complete tasks i and j by $[t_4 - t_2]_n + [t_4 - t_3]_n$ and the times to complete all other jobs remain unchanged.

Case 2: The time to complete tasks i and j in a job arriving in periods t_1, t_2, t_4 and t_5 will be $[t_3 - t_1]_n$, $[t_3 - t_2]_n$, $[t_6 - t_4]_n$, and $[t_6 - t_5]_n$ respectively. Now if tasks i and j are scheduled in an alternating fashion, these flowtimes become $[t_2 - t_1]_n$, $[t_3 - t_2]_n$, $[t_5 - t_4]_n$, and $[t_6 - t_5]_n$ which again is an improvement since all other flowtimes remain unchanged.

Consequently, if either Case 1 or Case 2 occurs, the flowtime will never increase and may be improved by the switches. If however, neither Case 1 nor Case 2 exists, then the tasks are being scheduled in an alternating fashion with at most one occurrence of the same task being offered consecutively. This implies that either both tasks are offered the same number of times or that if the less frequently scheduled task is scheduled k times, the other task will be scheduled k + 1 times.

As a result of Theorem 9.1, tasks are scheduled either $\left\lceil \frac{np}{\ell} \right\rceil$ or $\left\lceil \frac{np}{\ell} \right\rceil + 1$ times. If both sets are nonempty, the set of tasks that are scheduled $\left\lceil \frac{np}{\ell} \right\rceil$ times will be called bottleneck tasks. The other set of tasks that are scheduled $\left\lceil \frac{np}{\ell} \right\rceil + 1$ times will be called nonbottleneck tasks.

The nonbottleneck tasks may be scheduled in such a way that they will never cause delay; however, in some instances it is impossible to schedule the bottleneck courses such that they are not delay tasks.

Let the number of times each bottleneck is scheduled be $c = \left\lceil \frac{np}{\ell} \right\rceil$. Consider the simple problem of scheduling one bottleneck task independently of all other tasks. If the delay caused by this bottleneck task is to be minimized, then the problem is just to schedule c task in n time periods, which is the clock problem. Now consider the relaxation of the EPPS problem designated by EPPS′.

EPPS′: Given that there are an infinite number of processors available each period, determine a schedule for b different bottleneck tasks—each of which must be scheduled c times in n periods, such that delay is minimized.

EPPS′ is a relaxation of EPPS in two ways. First, the parameter p is removed from the problem since an infinite number of processors are assumed. Second, only bottleneck tasks are considered in the schedule. The second aspect of the relaxation will be unimportant if in the final schedule only bottleneck tasks cause delay.

Theorem 9.2: The optimal solution to EPPS′ problem is to schedule each set of b tasks in the periods $\{0, \left\lceil \frac{n}{c} \right\rceil, \left\lceil \frac{2n}{c} \right\rceil, \ldots \left\lceil \frac{(c-1)n}{c} \right\rceil\}$.

Proof: The total delay caused by the schedule given in the theorem is exactly equal to the delay of scheduling one task c times in n periods as given by the clock solution. Consequently, the schedule must be optimal.

The ESSP′ schedule will not be feasible to ESSP if b > p. The solution to EPPS′ does, however, give a lower bound for the delay time in the ESSP problem. Consider an example in which n = 7, p = 4, and ℓ = 11. The ESSP solution is indicated in Figure 9.8.

Periods						
0	1	2	3	4	5	6
C			G			
H			H			
I			I			
J			J			
K			K			

Figure 9.8. ESSP′ Solution.

The number of bottleneck tasks is $b = \ell - [np]_\ell = 5$, which is greater than p. Consequently, the schedule is not feasible to EPPS. Define a new problem EPPS′′ that is a restriction on EPPS′ and a relaxation of EPPS.

EPPS′′: Given that there are p processors available each period, determine a schedule for b different bottleneck tasks, each of which must be scheduled c times in n periods, such that total delay is minimized.

EPPS′′ is a relaxation of EPPS because it considers only the placement of bottleneck tasks without regard to possible delays that might be caused by nonbottleneck courses. EPPS′′ is a restriction on EPPS because only p processors are available each period. Therefore, the following relation exists:

$$\text{Delay (EPPS)} \geq \text{Delay (EPPS′′)} \geq \text{Delay (EPPS′)}$$

C. BOUNDS FOR THE EMPTY PRECEDENCE PROBLEM AND RELATED PROBLEMS

To calculate the bound for EPPS'' takes some insight into the nature of delays. If a delay $d > 0$ exists in some period i, then it follows that the delay in period $[i + 1]_n$ must be at least $d - 1$. The delay (d_i) for a job arriving in period i is the flowtime (f_i) minus the minimum possible flowtime $\left\langle \frac{\ell}{p} \right\rangle$. Consider an example in which $\left\langle \frac{\ell}{p} \right\rangle = 2$ and $n = 10$. A possible schedule for one task is given in Figure 9.9 where d_i^A = the delay caused by task A in period i.

	0	1	2	3	4	5	6	7	8	9
Schedule	A			A			A			
d_i^A	0	1	0	0	1	0	0	2	1	0

Figure 9.9. Delays for Task A.

In period 7, there is a delay of 2 periods because a job arriving in period 7 must wait until period 0 to have task A executed. The flowtime is 4 periods; however, the minimum flowtime is 2 periods. Therefore, the job encounters a delay of 2 periods.

Now in the same schedule, task B is isolated; and the delays are given in Figure 9.10.

	Periods									
	0	1	2	3	4	5	6	7	8	9
Schedule		B			B			B		
d_i^B	0	0	1	0	0	1	0	0	2	1

Figure 9.10. Delays for Task B.

If the two tasks A and B are considered together, a job arriving in period 8 would be delayed 1 period, waiting for task A to be executed, and 2 periods waiting for task B to be executed. The job's delay would be the larger delay or a delay of 2 periods. The method of calculating delay now becomes obvious. Consider each task j in the schedule independently and calculate the delay for each time period, d_i^j. The delay for each period d_i can then be calculated by:

$$d_i = \underset{\text{all } j}{\text{Max}}\ d_i^j$$

The total delay D is then given by

$$D = \sum_{i=0}^{n-1} d_i$$

Consider the following example in which $\ell = 17$, $p = 3$, and $n = 15$. The minimum flowtime is $\left\langle \frac{\ell}{p} \right\rangle = 6$. A possible schedule for this problem is given in Figure 9.11.

Periods

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	D	G	J	M	P	B	E	H	K	N	Q	C	F	I
B	E	H	K	N	Q	C	F	I	L	O	A	D	G	J
C	F	I	L	O	A	D	G	J	M	P	B	E	H	K

Figure 9.11. Example Schedule I

The delays for each task considered independently are shown in Table 9.1. In this example, D = 14.

Table 9.1. Delays for Individual Tasks for Example Schedule I

	Periods														
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
d_i^j, j=A...K	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
d_iL	0	0	0	0	0	0	0	0	0	0	3	2	1	0	0
d_iM	0	0	0	0	0	0	0	0	0	0	4	3	2	1	0
d_iN	0	0	0	0	0	0	0	0	0	0	0	3	2	1	0
d_iO	0	0	0	0	0	0	0	0	0	0	0	3	2	1	0
d_iP	0	0	0	0	0	0	0	0	0	0	0	4	3	2	1
d_iQ	0	0	0	0	0	0	0	0	0	0	0	0	3	2	0
d_i	0	0	0	0	0	0	0	0	0	0	4	4	3	2	1

Another possible schedule for the same problem is given in Figure 9.12 with the associated delays given in Table 9.2, and D = 7.

Periods

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
L	O	A	D	G	J	B	L	O	E	H	K	C	F	I
M	P	B	E	H	K	C	M	P	F	I	A	D	G	J
N	Q	C	F	I	A	D	N	Q	G	J	B	E	H	K

Figure 9.12. Example Schedule II.

Table 9.2. Delays for Individual Tasks for Example Schedule II

	Periods														
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
d_i^j, j=A...K	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
d_iL	0	1	0	0	0	0	0	0	2	1	0	0	0	0	0
d_iM	0	1	0	0	0	0	0	0	2	1	0	0	0	0	0
d_iN	0	1	0	0	0	0	0	0	2	1	0	0	0	0	0
d_iO	0	0	1	0	0	0	0	0	0	2	1	0	0	0	0
d_iP	0	0	1	0	0	0	0	0	0	2	1	0	0	0	0
d_iQ	0	0	1	0	0	0	0	0	0	2	1	0	0	0	0
d_i	0	1	1	0	0	0	0	0	2	2	1	0	0	0	0

In both schedules, tasks A through K are offered $\left[\frac{np}{\ell}\right] + 1$ times and are classified as nonbottleneck tasks and are all nondelay tasks. The tasks L through Q are bottleneck tasks because they are scheduled only $\left[\frac{np}{\ell}\right]$ times. In both examples, all bottleneck tasks are delay tasks.

To obtain a bound for EPPS'', only the bottleneck tasks need be considered. The number of bottleneck tasks b is given by

$$b = \ell - [np]_\ell$$

since $[np]_\ell$ is the number of nonbottleneck tasks. The number of times that each bottleneck task is offered is

$$c = \left[\frac{np}{\ell}\right]$$

Now in scheduling each individual bottleneck task c times, the minimum total delay due to that task is achieved by using the clock solution. The clock solution has gaps of at most two sizes:

$$g_1 = \left\lceil \frac{n-c}{c} \right\rceil + 1$$

and

$$g_2 = \left\lceil \frac{n-c}{c} \right\rceil$$

Referring back to Figure 9.9, p. 94, a job arriving in a period immediately after the bottleneck task A is scheduled will require either g_1 or g_2 periods to complete task A. Since the minimum flowtime for the entire job is $\left\langle \frac{\ell}{p} \right\rangle$, then the delay due to the bottleneck course A is either

$$q = \left\lceil \frac{n-c}{c} \right\rceil - \left\langle \frac{\ell}{p} \right\rangle + 1$$

or

$$s = \left\lceil \frac{n-c}{c} \right\rceil - \left\langle \frac{\ell}{p} \right\rangle$$

for a job arriving in a period immediately after that task is scheduled.

In the clock solution, the number of gaps of length g_1 was found to be

$$b - m \left\lceil \frac{b}{m} \right\rceil$$

Consequently, the number of gaps of length g_2 would be

$$m - (b - m \left\lceil \frac{b}{m} \right\rceil).$$

Now in the empty precedence problem $b = n - c$ and $m = c$. So after making the substitution, the number of delays of time q is given by

$$N_q = n - c(1 + \left\lceil \frac{n-c}{c} \right\rceil)$$

and the number of delays of duration s is

$$N_s = c(2 + \left\lceil \frac{n-c}{c} \right\rceil) - n.$$

So the bound on total delay for scheduling one bottleneck task is

$$N_q \{q + (q-1) + (q-2) + \ldots + 1\} + N_s \{s + (s-1) + (s-2) + \ldots + 1\}$$

This bound is also the bound for EPPS since all bottleneck tasks are scheduled in the same period for that problem.

In EPPS'', however, if $b > p$, it is not possible to schedule all bottleneck tasks in the same period. In fact, it will require $\left\langle \frac{b}{p} \right\rangle$ periods to schedule a complete set of bottleneck tasks. This implies that there must be at least $N_q \left\langle \frac{b}{p} \right\rangle$ periods with a delay of q and $N_s \left\langle \frac{b}{p} \right\rangle$ periods with a delay of s. Therefore, a better bound in EPPS'' is given by

$$N_q \{\left\langle \frac{b}{p} \right\rangle q + (q-1) + (q-2) + \ldots + 1\} + N_s \{\left\langle \frac{b}{p} \right\rangle s + (s-1) + (s-2) + \ldots + 1\}$$

It will be shown later in Section D that an algorithm realizes the bound and is hence an optimal algorithm.

In the previous example, where $n = 15$, $\ell = 17$, and $p = 3$, the total delay for Schedule II was 7. Substituting these values in the appropriate formulas gives

$$c = \left\lfloor \frac{np}{\ell} \right\rfloor = 2$$

$$q = \left\lfloor \frac{n-c}{c} \right\rfloor - \left\langle \frac{\ell}{p} \right\rangle + 1 = 2$$

$$s = \left\lfloor \frac{n-c}{c} \right\rfloor - \left\langle \frac{\ell}{p} \right\rangle = 1$$

$$N_q = n - c\left(1 + \left\lfloor \frac{n-c}{c} \right\rfloor\right) = 1$$

$$N_s = c\left(2 + \left\lfloor \frac{n-c}{c} \right\rfloor\right) - n = 1$$

$$b = \ell = [np]_\ell = 6.$$

The bound then is given by

$$1 \cdot \left\{\left\langle \frac{6}{3} \right\rangle 2 + (2-1)\right\} + 1 \cdot \left\{\left\langle \frac{6}{3} \right\rangle \cdot 1\right\} = 7.$$

Since the bound is achieved by Schedule II, it is a tight bound for both EPPS'' and EPPS.

D. ALGORITHMS FOR THE EMPTY PRECEDENCE PROBLEM AND RELATED PROBLEMS

Algorithm for the EPPS'':

$$\text{Let } S = \left\{0, \left\lceil \frac{n}{c} \right\rceil, \left\lceil \frac{2n}{c} \right\rceil, \ldots, \left\lceil \frac{(c-1)n}{c} \right\rceil\right\}$$

Step 1: Form an ordered list B of all bottleneck tasks.

Step 2: For each period $s \in S$ schedule all the bottleneck tasks in order from the top of list B starting in period s and continuing into the next period only when the current period is full.

For example, if n = 6, p = 4, and ℓ = 10, then there are 6 bottleneck tasks and S = {0,3}. The schedule is given in Figure 9.13.

Periods					
0	1	2	3	4	5
E	I		E	I	
F	J		F	J	
G			G		
H			H		

Figure 9.13. Schedule for EPPS'' from the Algorithm.

Theorem 9.2: The algorithm for the EPPS'' is an optimal algorithm.

Proof: The set S is a clock solution for scheduling c tasks in n time periods.

Case 1. $b \leq p$. The algorithm constructs a schedule that has the same delay as the clock solution and is therefore optimal.

Case 2. $b > p$. The number of time periods required by the algorithm to schedule the set B of bottleneck task is $\left\langle \frac{b}{p} \right\rangle$. Since each bottleneck task is scheduled by the clock solution, the total delay for the schedule is the same as the bound calculated for EPPS'' and is therefore optimal.

Since the EPPS'' ignores nonbottleneck tasks, it is not clear how to insert them. If they can be inserted in a way such that they are never delay tasks, then the same bound will be achieved and the problem is solved.

Consider an example where $n = 7$, $\ell = 5$, and $p = 3$. From these parameters $c = 4$ and $s = \{0,1,3,5\}$. The optimal solution to the EPPS'' is given in Figure 9.14.

	Periods						
	0	1	2	3	4	5	6
	B	E	D	B	E	B	E
	C	B	E	C		C	
	D	C		D		D	
d_i	0	0	0	0	0	0	0

Figure 9.14. Example Schedule for EPPS''.

If the nonbottleneck task A is inserted in the available time periods, the solution is given in Figure 9.15.

In this schedule, the nonbottleneck task A is a delay task. The difficulty in using the EPPS'' algorithm to solve EPPS is that no allowance is made for placing the nonbottleneck task while placing the bottleneck tasks. The algorithm for EPPS'' must be modified to assure

that in every set of $\left\langle \frac{\ell}{p} \right\rangle$ consecutive periods, all nonbottleneck tasks can be scheduled.

Periods

	0	1	2	3	4	5	6
	B	E	D	B	E	B	E
	C	B	E	C	(A)	C	(A)
	D	C	(A)	D		D	
d_i^A	1	0	0	0	0	0	0

Figure 9.15. Example Insertion of Nonbottleneck Tasks.

Algorithm for EPPS:

Initialization

Step 1: Set $s = \left\{0, \left\lfloor \frac{n}{c} \right\rfloor, \left\lfloor \frac{2n}{c} \right\rfloor, \ldots, \left\lfloor \frac{(c-1)n}{c} \right\rfloor \right\}$.

Step 2: Form an ordered list B of the bottleneck tasks.

Step 3: Form a circular ordered list $\overline{B}$ of the nonbottleneck tasks.

Loop

Step 4: Schedule the complete set of bottleneck tasks from the top of list B starting in the current period and continuing into future periods only when periods become full.

Step 5: Schedule a complete set of nonbottleneck tasks continuing from the last task scheduled from list $\overline{B}$ starting in the current period and continuing into future periods only when periods become full.

Step 6: If the current period is an element of S, go to step 4; otherwise continue scheduling nonbottleneck tasks from list $\overline{B}$ until either a period in S is encountered in which case go to step 4

or

all periods have been filled in which case stop.

Consider an example where n = 5, p = 3, and ℓ = 6. The bottleneck tasks then are C, D, E, F, and the nonbottleneck tasks A and B; S = {0,1,3}. The schedule generated by the EPPS algorithm is given in Figure 9.16.

	0	1	2	3	4
	C	A	E	C	A
	D	B	F	D	B
	E	C	A	E	
	F	D	B	F	
d_i	0	0	0	0	0

Figure 9.16. Example Schedule Using EPPS Algorithm.

The only difference between the algorithm for EPPS and the optimal solution for EPPS´´ is that the algorithm for EPPS shifts the bottleneck tasks to assure that between each set of bottleneck tasks is a complete set of nonbottleneck tasks. For example, in Figure 9.17, the solution to the EPPS´´ is given where n = 5, p = 3, and ℓ = 6.

Periods

0	1	2	3	4
C	C		C	
D	D		D	
E	E		E	
F	F		F	

Figure 9.17. Optimal Schedule for EPPS''.

To convert this solution to the solution that the algorithm would generate for the EPPS, the bottleneck tasks in period 1 must be shifted to allow for a complete set of nonbottleneck tasks between the two sets of bottleneck tasks. This shift will generate the solution that was given by the algorithm in Figure 9.16.

Theorem 9.3: The algorithm for the EPPS is an optimal algorithm.

Proof: The algorithm constructs schedules such that every set of $\left\langle \frac{\ell}{p} \right\rangle$ consecutive time periods contains a complete set of nonbottleneck tasks. Consequently, no nonbottleneck tasks will cause delay. Therefore, only the delays caused by bottleneck tasks need be considered.

Since EPPS is a relaxation of EPPS, it suffices to show that the schedule generated by the algorithm has the same delay as the EPPS bottleneck-task-only schedule. This can be done by showing that the shifts will never cause an increase in delay.

The only way to increase delay is to increase the number of time periods between bottleneck tasks. In the algorithm, the number of time periods between bottleneck tasks is increased only when they are

scheduled so close together that a complete set of nonbottleneck tasks cannot be scheduled between them. Before the shift, the schedule structure is as shown in Figure 9.18 where B is the complete set of bottleneck tasks in some specified order, $\overline{B}$ is the complete set of nonbottleneck tasks in some specified order, and $\overline{B}'$ is a proper subset of $\overline{B}$.

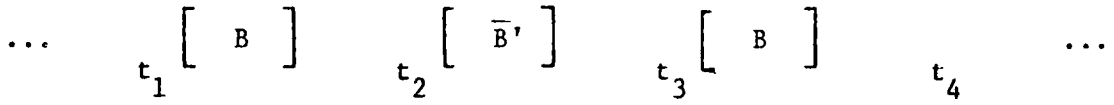

Figure 9.18. The Unshifted Schedule.

This is the situation that exists in Figure 9.17, since between the complete set of bottleneck tasks in period 0 and the complete set of bottleneck tasks in period 1, there are no nonbottleneck tasks or $\overline{B}' = \emptyset$.

After the shift, the structure is as shown in Figure 9.19.

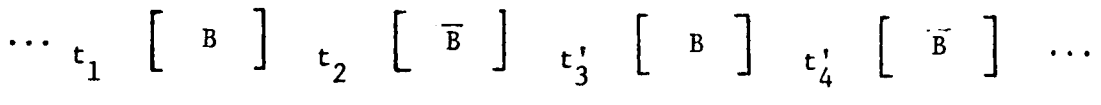

Figure 9.19. The Shifted Schedule.

This corresponds to Figure 9.16 where the bottleneck task contained completely in period 1 have been shifted to periods 1 and 2 to allow for the insertion of the complete set of nonbottleneck tasks (A,B).

The only jobs that could possibly be delayed by the shift are those arriving in periods t_1 through t_3. However, any job arriving in these

periods will have zero delay because there will be processors set up to execute all the tasks as quickly as possible. Therefore, the shift will never increase delay, and the theorem is established.

CHAPTER 10

AREAS FOR FUTURE RESEARCH

A. INTRODUCTION

With the exception of this work, periodic scheduling is essentially an unstudied area of scheduling theory. Consequently, one could take all the results in classical scheduling theory and study their counterparts in a periodic scheduling context. For example, there are many other performance measures and precedence structures to be considered. However, this work did motivate some immediate areas for future research. It is these problems that will be addressed in subsequent sections of this chapter.

In addition to periodic scheduling theory, the simulation model also contains some areas for improvement. These have to do primarily with programming changes to make the simulation model resemble the real academic situation more closely. The last section in this chapter will address these possible changes to the simulation program.

B. WEIGHTED CLOCK PROBLEM

A solution to the weighted clock problem was not obtained in this work. By a solution is meant a polynomial algorithm. However, some of the difficulties in deriving an algorithm were demonstrated in Chapter 7. Nevertheless, even though the evidence strongly suggests that no such algorithm is possible, this fact was not proved. It may be that for some

special case, for example, zero-one weights, an algorithm is possible. If no algorithm can be found, effort should be given to proving the problem NP-complete.

The importance of determining whether the weighted clock problem can be solved should be obvious. In the Serial Periodic Scheduling Problem (SPSP) in Chapter 8, only the unweighted case was considered. The optimal algorithm relied heavily on the unweighted clock problem solution. The same was true for the Empty Precedence Periodic Scheduling Problem (EPPS) in Chapter 9. Any attempt to find an algorithm for the weighted case of either the SPSP or EPPS should be delayed until the weighted clock problem is either solved or proved to be NP-complete.

Also, if the arrival function is periodic, this can be modeled as a weighted problem. Consequently, being able to solve the weighted clock problem would open many new problems for consideration. However, showing this problem to be NP-complete would imply the NP-completeness of many more complicated periodic scheduling problems for which the weighted clock problem is a special case.

C. EMPTY PRECEDENCE PERIODIC SCHEDULING PROBLEM WITH m > p

In Chapter 9, only the special case of the EPPS where the number of processors (m) equaled the number of tasks that jobs could have executing simultaneously (p) was considered. As was explained, this also solves the problem where p > m, since the number of processors sets a limit on the number of tasks in each job that can be executing simultaneously.

However, the case where $p < m$ was not addressed. This situation is not too important in machine-type situations, but it is very important in the academic scheduling problem. If m is the number of courses that are offered each period, and p is the maximum number of courses that each student can take per period, then the usual situation is $p < m$.

The major difficulty in the $p < m$ case is demonstrated in the following example. Consider the schedule shown in Figure 10.1

	Periods						
0	1	2	3	4	5	6	7
A	D	G	C	F	B	E	A
B	E	A	D	G	C	F	B
C	F	B	E	A	D	G	C

Figure 10.1. Example Schedule for the Case where $m < p$.

Suppose that a student enters in period 0 and can take only two courses per period ($p = 2$). A possible schedule for the student is given in Figure 10.2.

Periods				
0	1	2	3	4
A	D	G	C	F
B	E			

Figure 10.2. Example Student Schedule.

The difficulty comes from the fact that now the student has a choice as to which courses to take in each period. For example, in period 0, courses A, B, and C are offered. Of this set, the student may choose at most two courses. In all the other problems considered in this work, there was never a choice. In the serial problem, the precedence structure dictated which course must be taken. In the empty case where m = p, the student would take all courses that had not already been taken. Again, no decision was required. Now, however, the problem facing the student is a hard one. In fact, the only algorithm for the student's problem at this point appears to be enumeration. An optimal schedule for the example schedule given in Figure 10.1 is shown in Figure 10.3.

Periods

0	1	2	3
A	D	G	C
B	F		E

Figure 10.3. Optimal Student Schedule

This brings up an interesting question. Suppose that it can be shown that the problem facing the student for the situation is NP-complete. Does this imply anything about the complexity of the problem of scheduling the courses to time periods. In other words, if it is known that the student entering in each period will always follow an optimal schedule, is it possible to schedule the courses such that total

delay is minimized without actually knowing the individual student's schedule?

There is another variation of the same problem. Suppose that each student follows some known algorithm (not necessarily optimal). The problem is to determine the optimal schedule given that each student follows this algorithm. This problem has the advantage that the student's algorithm may be polynomial, thus perhaps making the course scheduling problem easier.

This type of problem is not restricted to scheduling problems. Consider the following analogous problem involving a traveling salesman. A salesman will follow some known algorithm in determining his route. The problem is to design an optimal network for the salesman to follow. For example, suppose there are many alternative arcs that can be constructed between each pair of nodes at different costs. The problem is to construct the network, within some cost constraint, such that the salesman's travel distance is minimized.

D. SOME ADDITIONAL PROBLEMS

There are several other precedence structures for which polynomial algorithms may exist. The first is called a parallel series precedence structure and is depicted in Figure 10.4.

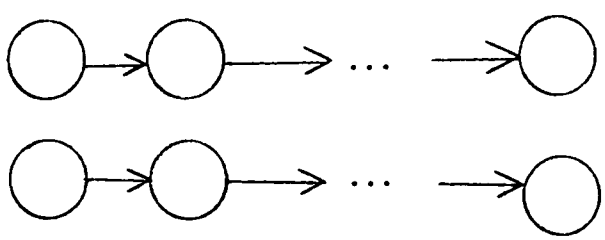

Figure 10.4. Parallel Series Precedence Structure.

The other type of precedence structure that should be considered is a tree precedence structure. Neither of these precedence structures has been studied sufficiently. However, enough time was spent to ensure that a solution is not trivial.

Besides looking at different precedence structures, there are other variations to be considered. The only performance measure considered was delay or equivalently flowtime. For example, tardiness or number of tardy jobs makes the problems considerably different. In all periodic scheduling problems, it was assumed that each processor once set up to execute a particular task could execute an unlimited number of that type of task simultaneously. This is a very unrealistic assumption. In the academic case, this limit implies that class sizes are infinite. In the machine-type situation, usually a processor can only execute one task at a time. Therefore, in both situations, the infinite processor capacity is rather simplistic. Removing it, however, at this point appears to make the problem intractable.

Also, all the problems considered have tasks with unit execution time because of the academic motivation for the problem. Future research, however, should address the problem of unequal execution time.

E. REFINEMENTS IN THE SIMULATION

In using the simulation, to date only one major deficiency has been found. The simulation does not take into account the differences between the summer quarter and other quarters. In many schools using the quarter system, the summer quarter is divided into two sessions.

This presents a difficult situation. It is possible for a student to take a course in the second session that has as a prerequisite a course that was taken in the first session. Currently, the simulation program will not allow this situation because of a prerequisite conflict.

Also, because the simulation was originally designed for a graduate program, in which students usually attend year round, the simulation does not allow students to attend fall, winter, and spring quarters and skip the summer quarter.

Unfortunately, because of the design of the simulation program, these changes will be difficult to implement. Therefore, installation of these changes has been delayed until the usefulness of the original version is determined.

REFERENCES

REFERENCES

1. Adolphson, D. L., "Single Machine Job Sequencing with Precedence Constraints." SIAM J. of Computing, Vol. 6, No. 1 (March, 1977), pp. 40-54.

2. Aho, A. V., M. R. Garey, and J. D. Ullman, "The Transitive Reduction of a Directed Graph," SIAM J. of Computing, Vol. 1, No. 2 (June, 1972), pp. 131-137.

3. Baker, Kenneth R., Introduction to Sequencing and Scheduling, John Wiley & Sons, New York, 1974.

4. Bruno, J. E., E. G. Coffman, and R. Sethi, "Scheduling Independent Tasks to Reduce Mean Finishing Time," Communication of ACM, Vol. 17, No. 7 (1974), pp. 382-387.

5. Coffman, E. G., Computer and Job Shop Scheduling Theory, John Wiley & Sons, New York, 1976.

6. Coffman, E. G., and R. L. Graham, "Optimal Scheduling for Two-Processor Systems," Acta Informatica, Vol. 1, No. 3 (1972), pp. 200-213.

7. Conway, R. W., W. L. Maxwell, and L. W. Miller, Theory of Scheduling, Addison-Wesley, Reading, Massachusetts, 1967.

8. Cook, Stephen A., "The Complexity of Theorem-Proving Procedures," Proc. 3rd ACM Symp. on Theory of Computing (1971), pp. 151-158.

9. Garey, M. R., and D. S. Johnson, "Complexity Results for Multi-processor Scheduling under Resource Constraints," SIAM J. of Computing, Vol. 4, No. 4 (December, 1975), pp. 397-411.

10. Garey, M. R., and D. S. Johnson, Computers and Intractability—A Guide to the Theory of NP-Completeness, W. H. Freeman and Company, San Francisco, 1979.

11. Garey, M. R., D. S. Johnson, and Ravi Sethi, "The Complexity of Flowshop and Jobshop Scheduling," Math. of Oper. Res., Vol. 1, No. 2 (May, 1976), pp. 117-129.

12. Horn, W. A., "Single-Machine Job Sequencing with Treelike Precedence Ordering and Linear Delay Penalties," SIAM J. of Appl. Math., Vol. 23 (1972), pp. 189-202.

13. Horn, W. A., "Minimizing Average Flowtime with Parallel Machines," Oper. Res., Vol. 21 (1973), pp. 846-847.

14. Hu, T. C., "Parallel Sequencing and Assembly Line Problems," Oper. Res., Vol. 9, No. 6 (1961), pp. 841-848.

15. Johnson, S. M., "Optimal Two- and Three-Stage Production Schedules with Setup Times Included," Naval Research Logistics Quarterly, Vol. 1, No. 1 (March, 1954).

16. Lageweg, B. J., J. K. Lenstra, and A. H. G. Rinnooy Kan, "A General Bounding Scheme for the Permutation Flow-Shop Problem, Oper. Res., Vol. 26, No. 1 (January-February, 1978), pp. 53-67.

17. Lawler, Eugene L., "A 'Pseudo-polynomial' Algorithm for Sequencing Jobs to Minimize Total Tardiness," Ann. of Disc. Math., Vol. 1 (1977), pp. 331-342.

18. Lawler, E. L., "Sequencing Jobs to Minimize Total Weighted Completion Time Subject to Precedence Constraints," Ann. of Disc. Math., Vol. 2 (1978), pp. 75-90.

19. Lenstra, J. K., A. H. G. Rinnooy Kan, and P. Brucker, "Complexity of Machine Scheduling Problems," Ann. of Disc. Math., Vol. 1 (1977), pp. 343-362.

20. Lenstra, J. K., and A. H. G. Rinnooy Kan, "Complexity of Scheduling under Precedence Constraints," Oper. Res., Vol. 26, No. 1 (January-February, 1978), pp. 22-35.

21. Muntz, R. R., and E. G. Coffman, "Optimal Preemptive Scheduling on Two-Processor Systems," IEEE Trans. on Comput., Vol. 18, No. 11 (November, 1969).

22. Rinnooy Kan, A. H. G., Machine Scheduling Problems Classification, Complexity and Computations, Martinus Nijhoff, The Hague, 1976.

23. Sethi, R., "On the Complexity of Mean Flow Time Scheduling," Math. of Oper. Res., Vol. 2, No. 4 (November, 1977), pp. 320-330.

24. Smith, W. E., "Various Optimizers for Single Stage Production," Naval Research Logistics Quarterly, Vol. 3 (1956), pp. 59-66.

25. Ullman, J. D., "Polynomial Complete Scheduling Problems," Operating Systems Review, Vol. 7, No. 4 (1973), pp. 96-101.

26. Ullman, J. D., "NP-Complete Scheduling Problems," J. of Comput. and Sys. Sci., Vol. 10 (1975), pp. 384-393.

VITA

William John Plotnicki was born in Knoxville, Tennessee, on June 12, 1949. He attended both elementary and high school in that city. In September, 1967, he enrolled in The University of Tennessee, Knoxville, from which he received his Bachelor of Science in Engineering Physics in March, 1972. He was valedictorian of the College of Engineering and voted the most outstanding Physics student.

He accepted a graduate assistantship from Colorado State University. In March, 1974, he received his MBA from that school. In September, 1974, he enrolled in the Management Science Program at The University of Tennessee, Knoxville, with a graduate teaching assistantship. He received his Doctor of Philosophy in August, 1980. He is currently an Assistant Professor in the Management Science and Information Systems Department at Colorado State University.

The author is a member of Operations Research Society of America, The Institute of Management Science, The Association for Computing Machinery, Beta Gamma Sigma and Phi Kappa Phi.