PB 214 612

STAN-CS-72-324 SU-SEL-72-057

Subproblem of the m x n Sequencing Problem

by

Henry Raymond Bauer

November 1972

Technical Report No. 48

This work was supported by the
National Science Foundation
under Grant GJ 1180

DIGITAL SYSTEMS LABORATORY
STANFORD ELECTRONICS LABORATORIES
STANFORD UNIVERSITY · STANFORD, CALIFORNIA

Reproduced by
NATIONAL TECHNICAL
INFORMATION SERVICE
U S Department of Commerce
Springfield VA 22151

BIBLIOGRAPHIC DATA SHEET	1. Report No. STAN-CS-72-324	2.	3. Recipient's Accession No. [illegible]
4. Title and Subtitle Subproblem of the m x n Sequencing Problem			5. Report Date November 1972
			6.
7. Author(s) Henry Raymond Bauer			8. Performing Organization Rept. No. STAN-CS-72-324
9. Performing Organization Name and Address Stanford University Computer Science Department Stanford, California 94305			10. Project/Task/Work Unit No.
			11. Contract/Grant No. GJ 1180
12. Sponsoring Organization Name and Address National Science Foundation Washington, D. c.			13. Type of Report & Period Covered technical
			14.
15. Supplementary Notes			
16. Abstracts (attached)			
17. Key Words and Document Analysis. 17a. Descriptors none listed			
17b. Identifiers/Open-Ended Terms			
17c. COSATI Field/Group			
18. Availability Statement Distribution Unlimited.	19. Security Class (This Report) UNCLASSIFIED		21. No. of Pages ~~116~~ 123
	20. Security Class (This Page) UNCLASSIFIED		22. Price $3.00

FORM NTIS-35 (REV. 3-72)

THIS FORM MAY BE REPRODUCED

USCOMM-DC 14952-P72

Subproblems of the m × n Sequencing Problem

ABSTRACT

Assume that a set of n tasks is to be scheduled on m processors. Each task is indivisible, and each processor may be concerned with only one task at a time. Then the m × n sequencing problem is to find the schedule in which the total completion time for all tasks is minimal. In addition, we seek an algorithmic solution which is efficient; that is, the computation must grow algebraically with the size of the problem rather than combinatorially.

The results presented concern three separate families of subproblems. The first problem is an extension of the problems of Hu, and Coffman and Graham. Here we develop an algorithm for the optimal sequencing of n 1-unit and 2-unit tasks with tree precedence on two processors.

The second family of problems concerns the sequencing of tasks on two processors where the tasks consist of chains of operations with known lengths. Furthermore, the operations of each task are to be performed alternately on the two processors. The new results, including knapsack solutions, are algorithms for tasks which consist of:

a. three operations with 1-unit and 2-unit lengths, or

b. three operations each of whose adjacent operations differ in length by 1-unit, or

c. three operations each of whose adjacent operations differ in length by k-units, or

d. three operations in which the first and last have identical lengths, or

e. four operations in which the first and last have identical lengths and

the second and third have identical lengths.

The algorithms for the third family of problems follow the work of Arthanari and Mukhopadhyay, and Szwarc. In our case, we treat m, $m \geq 4$, processors and tasks formed by chains of m operations. Each of the operations corresponds to each of the processors, in order. The results include:

a. a case in which the identical task order on adjacent processors yields an optimal schedule, and

b. the solution of a constrained four processor problem by solving n two processor problems, and

c. the reduction of a constrained four processor problem to the solution of a three processor problem.

Some systems programming problems from computer science have characteristics similar to our subproblems.

STAN-CS-72-324 SEL 72-057

SUBPROBLEM OF THE m x n SEQUENCING PROBLEM

by

Henry Raymond Bauer

Technical Report no. 48

November 1972

DIGITAL SYSTEMS LABORATORY

Department of Electrical Engineering Department of Computer Science

Stanford University

Stanford, California

This work was supported by the National Science Foundation under grant GJ 1180.

ib

Subproblems of the $m \times n$ Sequencing Problem

ABSTRACT

Assume that a set of n tasks is to be scheduled on m processors. Each task is indivisible, and each processor may be concerned with only one task at a time. Then the m × n sequencing problem is to find the schedule in which the total completion time for all tasks is minimal. In addition, we seek an algorithmic solution which is efficient; that is, the computation must grow algebraically with the size of the problem rather than combinatorially.

The results presented concern three separate families of subproblems. The first problem is an extension of the problems of Hu, and Coffman and Graham. Here we develop an algorithm for the optimal sequencing of n 1-unit and 2-unit tasks with tree precedence on two processors.

The second family of problems concerns the sequencing of tasks on two processors where the tasks consist of chains of operations with known lengths. Furthermore, the operations of each task are to be performed alternately on the two processors. The new results, including knapsack solutions, are algorithms for tasks which consist of:

a. three operations with 1-unit and 2-unit lengths, or

b. three operations each of whose adjacent operations differ in length by 1-unit, or

c. three operations each of whose adjacent operations differ in length by k-units, or

d. three operations in which the first and last have identical lengths, or

e. four operations in which the first and last have identical lengths and

the second and third have identical lengths.

The algorithms for the third family of problems follow the work of Arthanari and Mukhopadhyay, and Szwarc. In our case, we treat m, m ≥ 4, processors and tasks formed by chains of m operations. Each of the operations corresponds to each of the processors, in order. The results include:

a. a case in which the identical task order on adjacent processors yields an optimal schedule, and

b. the solution of a constrained four processor problem by solving n two processor problems, and

c. the reduction of a constrained four processor problem to the solution of a three processor problem.

Some systems programming problems from computer science have characteristics similar to our subproblems.

Acknowledgements

The preparation of a dissertation requires cooperation, encouragement, and criticism from many persons. This dissertation is no exception. I thank all those friends who have made my work at Stanford pleasurable and rewarding. In particular, there are several who deserve special recognition. Cne is my adviser, Professor Harold Stone, whose stimulating suggestions, constructive criticism, and profuse patience encouraged me to complete this project. Also I thank Professors Forest Baskett and Thomas Bredt for their careful readings of the manuscript.

My appreciation is alsc extended to Louise Hazlett and Joyce Vought who typed the final draft.

This research was supported in part by the National Science Foundation under grant number GJ-1180. Reproduction in whole or in part is permitted for any purpose of the United States Government.

TABLE OF CONTENTS

CHAPTER		PAGE
1.	THE m × n SEQUENCING PROBLEM	1
	1.1. The Problem Statement	1
	1.2. The Contents	5
2.	HISTORICAL PERSPECTIVE	6
	2.1. General Discussion	6
	2.2. Johnson's Result	7
	2.3. Identical Order on m Processors	10
	2.4. "Cutting the Longest Queue" Algorithms	12
	2.5. One Unit Tasks with Acyclic Procedure	16
	2.6. Two Observations	18
3.	ONE-UNIT AND TWO-UNIT TASKS	21
	3.1. The Problem Statement	21
	3.2. Development of the Solution to the Chain Precedence Problem	25
	3.2.1. Algorithm 3.1 -- Labelling	25
	3.2.2. Algorithm 3.2 -- Individual Level Scheduling	29
	3.2.3. Solution to the Chain Precedence Problem	32
	3.3. Solution to the Tree Precedence Problem	47
	3.3.1. Analysis of Algorithms 3.4 and 3.5	61
	3.4. A Solution to the Tree-Restricted Acyclic Precedence Problem	63
	3.5. A Look Ahead	65

4. SEGMENTED PROCESSOR SCHEDULING 66

4.1. The Segmented Scheduling Problem 66

4.2. Segmented Scheduling Problem 69

4.2.1. A Foundation for New Results 72

4.2.2. A Special Segmented Problem 74

4.2.3. A More General Core Problem 79

4.2.4. A Problem with a Knapsack Solution 86

4.2.5. Extension to the Four Stage Problem 91

4.3. Other Subproblems 93

5. THE FOUR PROCESSOR PROBLEM 94

5.1. The Four Processor Problem 94

5.2. Problem Definition 94

5.3. Restrictions on Permutations p and q 95

5.4. Extension of Szwarc's Results 96

5.5. In Summary 105

6. FUTURE DIRECTIONS 107

BIBLIOGRAPHY 109

LIST OF TABLES

3.1. Assignment of Forms 35

LIST OF FIGURES

		PAGE
1.1.	Gantt Chart Example	2
2.1.	Identical Order is Not Always Optimal	11
2.2.	Arthanari and Mukhopadhyay Notation	13
2.3.	Hu's "Cutting the Longest Queue" Algorithm	15
2.4.	Hu's Algorithm for Unequal Task Lengths	17
2.5.	Coffman and Graham Algorithm Example	19
3.1.	Example of Set G in Problem 3.1	22
3.2.	Example of Set G in Problem 3.2	22
3.3.	Example of Set G in Chain Precedence Problem	24
3.4.	Chains and Trees	26
3.5.	Example of the Labelling Algorithm	26
3.6.	Level Notation	28
3.7.	Use of Algorithm 3.2 in Example of Figure 3.4	30
3.8.	Form A, Form B, and Form C	36
3.9.	Modifications	38-39
3.10.	1-Unit Tasks in Algorithm 3.3	42
3.11.	Examples of Definitions	44
4.1.	Forms of the Special Segmented Problem	68
4.2.	No Solution to Problem 4.1 in Form II	70
4.3.	No Solution to Problem 4.1 in Form I	71
4.4.	Example of Algorithm 4.1	80
4.5.	Problem 4.3 and Algorithm 4.2	81
4.6.	Problem 4.4 and Algorithm 4.2	81
4.7.	First Form of Problem 4.5 Solution	87

4.8. Second Form of Problem 4.5 Solution 87

4.9. Example of Algorithm 4.3 89

5.1. Example of Theorem 5.1 101

5.2. Example of Theorem 5.2 102

Chapter 1

The m × n Sequencing Problem

A classic problem of operations research and management science is the optimal sequencing of n jobs on m processors. In computer science the area of systems programming again involves the same problem. In all cases the solution must be embodied in an efficient algorithm. The solutions of complicated sequencing problems often depend upon efficient algorithms for the less complex problems of scheduling. This work continues the development of these algorithms.

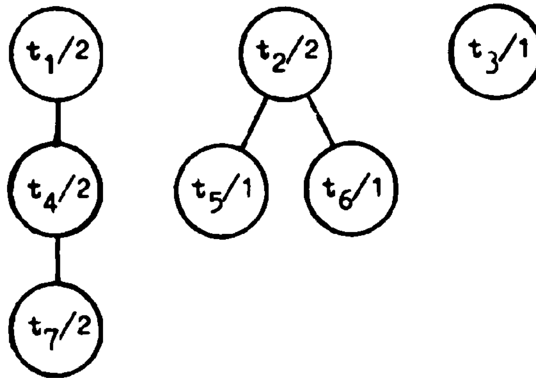
Since algorithms for the m × n sequencing problem exist, what are the efficient algorithms we seek?

Definition 1.1. An *efficient algorithm* produces an optimal solution to a problem using a computation whose size grows algebraically with the size of the problem.

First, efficient algorithms differ from those procedures that examine all possible solutions. These enumerative computations often grow combinatorially with the size of the problem. Second, efficient algorithms differ from those procedures that reduce an inherently combinatorial enumeration heuristically. Heuristic computations do not necessarily produce optimal solutions.

1.1. The Problem Statement

The general problem which concerns us is the *m × n sequencing problem*. Its definition below names the form of its solution a *schedule*. Authors differ, however, on the notation of their schedules. Here each schedule is denoted by a Gantt chart [Clark 1947]. As Figure 1.1 shows,



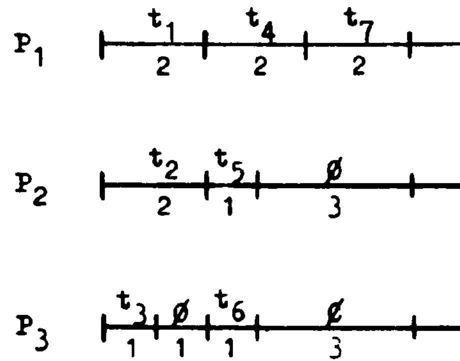


Figure 1.1. Gantt chart example

one of the set of horizontal, parallel time bars corresponds to each processor. Time intervals on each time bar are delimited by vertical bars. During a time interval a task may be executed on the specific processor. The task's name then appears above the time bar, and the task's length appears below the time bar. ϕ represents an idle time interval. In each problem an optimal schedule is a schedule in which all tasks complete execution in a minimum time. Definition 1.2 describes the general $m \times n$ sequencing problem.

Definition 1.2. The $m \times n$ Sequencing Problem

> A set of n tasks is to be scheduled on m processors. Each task is indivisible and may have to be processed before or after other tasks. Each processor may be concerned with only one task at a time. The $m \times n$ sequencing problem is to find the schedule in which the total completion time for all tasks is minimal.

This basic definition is frequently modified to produce a tractable problem. For example, the number of processors may have identical characteristics or may perform specialized operations. Alternatively, it may be desirable to find the minimum number of machines, m, required to attain the minimal total completion time of all tasks. Or the characteristics of the tasks may be varied so that, for example, they may be executed only in a special order.

The research results here concern three distinct classes of problems. The first class concerns 1-unit and 2-unit length tasks to be sequenced on two processors ($m = 2$). In one subproblem the tasks have at most one predecessor and at most one successor. A second subproblem concerns tasks with many predecessors but with at most one successor. Sets of these tasks are then maximally connected to similar sets of tasks in a third problem. The general problem of sequencing 1-unit and 2-unit tasks with an arbitrary number of predecessors and successors remains unsolved.

This first class of problems includes those treated by Hu [1961]

and by Coffman and Graham [1972]. Hu's problem concerns only 1-unit tasks with many predecessors but with at most one successor to be sequenced on m processors Coffman and Graham considered 1-unit tasks with an arbitrary number of predecessors and successors to be sequenced on two processors ($m = 2$). Our results maintain the $O(n)$ computational complexity established by Hu for his solution.

The second class of problems concerns a specific two processor sequencing problem. The work extends the results of Johnson [1954] and Bauer and Stone [1970]. In the latter work tasks consist of three operations (stages) to be performed on processor one, processor two, and processor one, respectively. The time interval for processing these tasks is given. Bauer and Stone succeed in isolating the difficult core of this problem. Here we present new results on the problem in which tasks consist of

a. three operations with 1-unit and 2-unit lengths, or

b. three operations each of whose adjacent operations differ in length by 1-unit, or

c. three operations each of whose adjacent operations differ in length by k-units, or

d. three operations in which the first and last have identical lengths, or

e. four operations in which the first and last have identical lengths and the second and third have identical lengths.

The third class of problems concerns tasks of varying length to be sequenced on four processors. The published results of Szwarc [1968], and Arthanari and Mukhopadhyay [1971] considered three processors. Our work extends these results to four processors and, in one case, to m processors. The results include

a. a case in which the identical task order on adjacent processors yields an optimal schedule, and

b. the solution of a constrained four processor problem by n applications of Johnson's method, and

c. the reduction of a constrained four processor problem to the solution of a three stage problem.

1.2. The Contents

In this dissertation Chapter 2 presents the history of the problem. Instead of repeating references to the many tangentially relevant papers, we cite more complete surveys of the literature. Only work which has influenced the present work directly is included here. In addition, we discuss the importance to computer science of this classical problem from operations research.

Chapter 3 presents the results concerning the first class of problems. The three problems appear together with their algorithmic solutions.

Chapter 4 defines the specific two processor sequencing problem of the second class of problems. Each subproblem appears individually with the discussion of its solution.

Chapter 5 extends the work of Szwarc, and Arthanari and Mukhopadhyay reviewed in Chapter 2. These results correspond to the third class of problems discussed above.

The last chapter contains a discussion of the significance of the work and the possible extensions in future research.

Chapter 2

Historical Perspective

2.1. General Discussion

Scheduling and sequencing research is a general name applied to numerous problems. Day and Hottenstein [1970] defined a schema for classifying sequencing problems. Using the Day-Hottenstein schema, we answer several questions.

What is the nature of job arrivals? In all cases we treat a batch with a fixed size. We do not treat the problem in which jobs continuously arrive satisfying some probability density function. Our problem restriction is reasonable for some computer systems. In such systems tasks may be accumulated with accurate time estimates. Then the assumption of a fixed, known-in-advance, batch size is correct. One example is a set of programs in a student environment to be scheduled for compilation, execution, and printer output. These tasks may be set aside and run as a large batch to improve the system performance by reducing overhead costs.

How many machines are involved? Each case concerns a multi-machine situation. In Chapters 3 and 4 the number of processors is limited to two ($m = 2$). In Chapter 5 four machines ($m = 4$) are discussed. However, we are primarily interested in the two processor problem. The results of Hu [1961] and Coffman and Graham [1972] indicate that two processor problems are perhaps the most amenable to efficient solution.

What is the nature of the job route? Here we treat two distinct situations. In Chapter 3 the tasks may be assigned to either processor. The order of the tasks is the only constraint. In Chapter 4 the order of the jobs is again constrained, and, in addition, each task must be assigned

to a specific processor.

Again several examples in computer science exhibit these restrictions. A complete job may consist of many tasks, one or several of which are to be completed before another may start. Tasks may be executed on any processor, but the order of execution is important. A second example involves processors dedicated to specific tasks. This case exists in a previous example when one processor compiles a program, a second executes the program, and a third prints the output. Alternatively, the input-output channel processor may perform the input, the central processor may perform the execution of the program, and the input-output channel processor, again, may then do the output.

The wealth of papers dealing with m $\times$ n sequencing is impressive. The scarcity of efficient algorithmic solutions is likewise remarkable. These facts attest to the relevance of the problem and to its difficulty. The reader may find several surveys of this subject in the literature [Bellman 1956, Conway, et. al. 1967, Day and Hottenstein 1970]. Here we use combinatorial approaches rather than solution methods using mathematical programming or heuristic programming.

Combinatorial solutions are those solutions which are based on finding the optimal permutation by changing from one task ordering to another. The objective is to find the optimal permutation but to avoid complete enumeration over the entire solution space. The families of papers discussed below have influenced research in this area considerably.

2.2. Johnson's Results

The first major results in the problem are by Johnson [1954]. Johnson considered the production schedule of n tasks with two operations each. The first operation is performed on the first machine, and the

second operation is performed on the second machine. There are only two machines. The second operation may not begin before the first operation is completed. Two of Johnson's results are Theorem 2.1 and Theorem 2.2, below. These theorems are the basis for the first efficient algorithms for the general sequencing problem.

Theorem 2.1. The order of the production sequence on two machines may be made the same without loss of time. [Johnson 1954]

Theorem 2.2. Johnson's Rule

Let tasks i, i = 1, 2,..., n, consist of the pair of operations a_i, b_i, where a_i, i = 1, 2,..., n, are the lengths of the operations to be processed on the first machine and b_i, i = 1, 2,..., n, are the lengths of the succeeding operations to be processed on the second machine. An optimal ordering is given by the rule:

Item j precedes item j+1 if

$$\min (a_j, b_{j+1}) < \min (a_{j+1}, b_j)$$

This ordering is unique except for ties. [Johnson 1954]

Johnson generalized Theorem 2.1 for n tasks, each with m operations to be performed on m machines, $m \geq 2$. Theorem 2.3 states the result which we use later.

Theorem 2.3. The order of the production sequence may be made the same on processor 1 and processor 2 and may be made the same on processor m-1 and processor m without loss of time. [Johnson 1954]

An additional Johnson result concerns n tasks with three operations each. The operations are assigned, respectively, to three processors (m = 3).

Theorem 2.4. Let tasks i, i = 1, 2,..., n, consist of the triplet of operations a_i, b_i, c_i where a_i, b_i, and c_i, i = 1, 2,..., n, are the lengths of the operations to be processed on machines 1, 2, and 3, respectively. Assume that all first operations are not less than any second operations, $\min_i a_i \geq \max_j b_j$. Task i precedes task j if

$$\min (a_i+b_i,\ c_j+b_j) < \min (a_j+b_j,\ c_i+b_i) \quad .$$

The only complete solution [Jackson 1956] for the general m × n sequencing problem for which the computational complexity is algebraic rather than exponential in n is for two processors (m = 2). Jackson produces Theorem 2.5. Later we use concepts from this result in our work.

Theorem 2.5. Let

{A} be the set of jobs with only one operation to be performed on machine one,

{B} be the set of jobs with only one operation to be performed on machine two,

{AB} be the set of jobs which have two operations, the first to be performed on machine one and the second on machine two,

and {BA} be the set of jobs which have two operations, the first to be performed on machine two and the second on machine one.

Then determine the sequence of tasks in {AB} and {BA} by Johnson's rule, and, using these orderings, assign the tasks to machine one and machine two as follows:

Machine One: tasks in {AB}, followed by tasks in {A}, followed by tasks in {BA}

Machine Two: tasks in {BA}, followed by tasks in {B}, followed by tasks in {AB}

where the order of tasks in {A} and {B} does not matter. [Jackson 1956]

Bauer and Stone [1970] used similar results for a somewhat different problem. Results of that research are in Chapter 4.

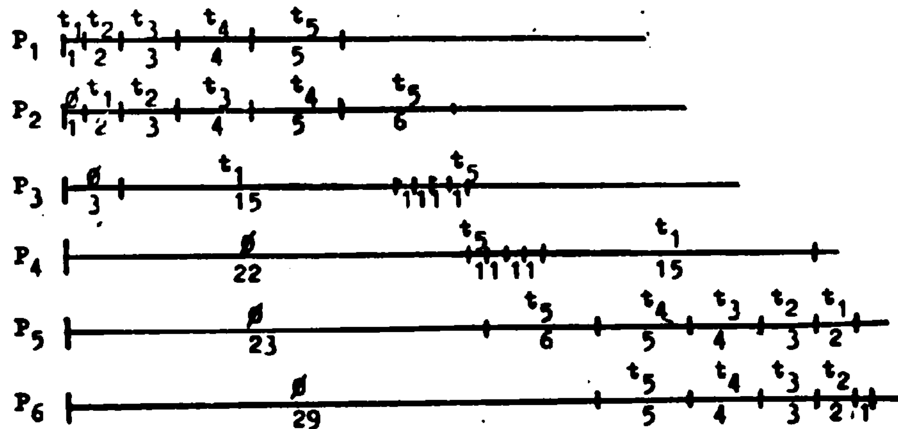
2.3. Identical Order on m Processors

A second family of results concerns the $m \times n$ sequencing problem with an identical processing order required on each machine. In general, the solution of this problem does not necessarily lead to the solution of the problem in which the processing order of tasks is not required to be identical. Figure 2.1 shows an example in which the optimal solution to the general $m \times n$ sequencing problem differs from the solution to the problem with identical task ordering. However, because of the complexity of improving the solution for the general case, the problem with identical task ordering is often deemed practical.

The principle results for the problem with identical task ordering are those of Dudek and Teuton [1964], Karush [1965], Smith and Dudek [1966, 1969], Szwarc [1968], and Arthanari and Mukhopadhyay [1971]. All the results have the characteristic of providing a decision rule for the ordering of tasks. The complexity of these rules is evidenced by half of the references listed being corrections of the other half.

Arthanari and Mukhopadhyay [1971] extended Szwarc's results [1968] to reduce the $3 \times n$ problem to repeated applications of Johnson's result, Theorem 2.2. The Arthanari and Mukhopadhyay problem is stated in Problem 2.1. No efficient algorithm is known for the $3 \times n$ sequencing problem without restrictions upon the size of the tasks. This method of problem reduction is the basis of work described in Chapter 5.

Optimal schedule — time = 44 units



processor / tasks	P_1	P_2	P_3	P_4	P_5	P_6
t_1	1	2	15	15	2	1
t_2	2	3	1	1	3	2
t_3	3	4	1	1	4	3
t_4	4	5	1	1	5	4
t_5	5	6	1	1	6	5

Minimal identical order schedule — time = 57 units

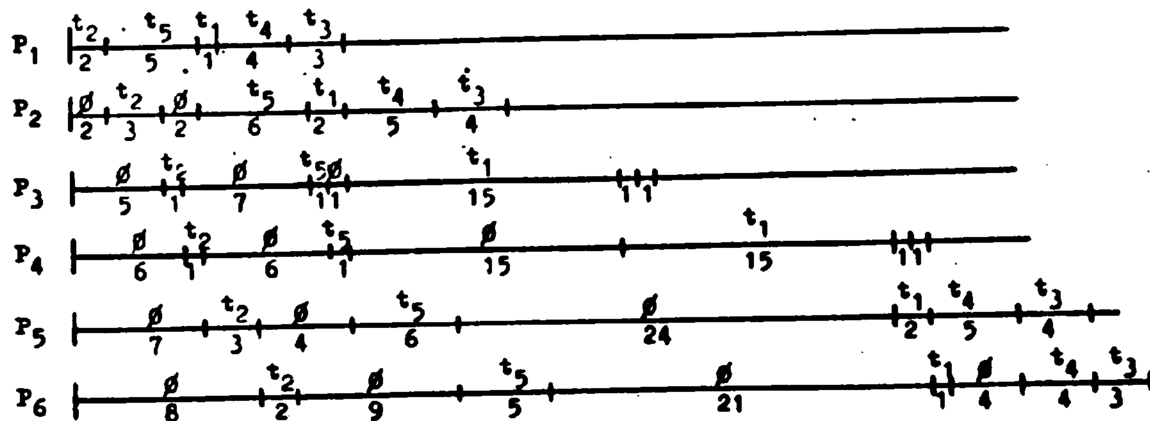


Figure 2.1. Identical order is not always optimal

Problem 2.1.

Find the optimal schedule for the $3 \times n$ sequencing problem in which each of the n tasks consists of three operations a_i, b_i, and c_i, $i = 1, 2, \ldots, n$. These operations are to be executed on processors 1, 2, and 3, respectively. Assume for all tasks that either

$$\max_{1 \le k \le n} c_k \le \min_{1 \le k \le n} b_k$$

or

$$\max_{1 \le k \le n} a_k \le \min_{1 \le k \le n} b_k$$

In the situation in which $\max_k a_k \le \min_k b_k$, they found that the schedule on machine two and machine three was most critical. Let some task i be the first task assigned, and let all other tasks on machine two and machine three be scheduled using Johnson's method from Theorem 2.2. Then call the idle time on the third processor I_i for the partial schedule obtained by Johnson's method for processors two and three. When all three processors are considered, the total idle time, D_i, on machine three is

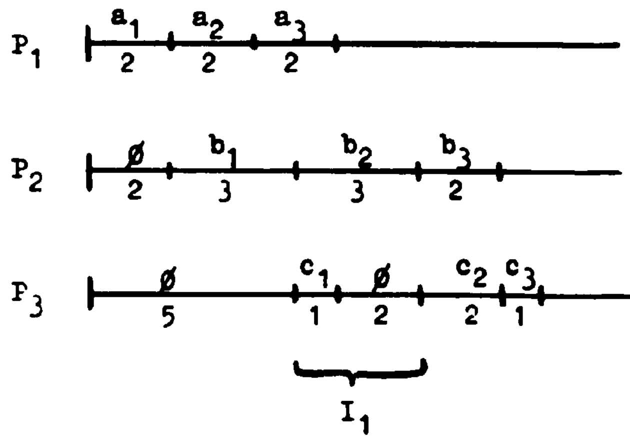
$$D_i = a_i + b_i - c_i + \max (c_i, I_i)$$

where task i is the first task executed. The solution then is simply to find the minimum value of D_i for all i, $i = 1, 2, \ldots, n$. The n applications of Johnson's algorithm will locate the optimal sequence. The situation for $\max_k c_k \le \min_k b_k$ has a similar solution. Figure 2.2 depicts the various quantities discussed above.

Extensions to these results appear in Chapter 4.

2.4. "Cutting the Longest Queue" Algorithms

A third family of sequencing problems concerns the $m \times n$ sequencing problem in which the processing order of the tasks is restricted. We recall several definitions from graph theory and then define the sequencing problems in these terms.



I_1 = 3 units

D_1 = 2 + 3 - 1 + max (1, 3) = 7 units

Figure 2.2. Arthanari and Mukhopadhyay notation

Definition 2.1. Given two tasks x and y, $x < y$ (read, x precedes y) if x must be processed before y may begin to be processed. Similarly, $x > y$ (read, x succeeds y) if y must be processed before x begins to be processed. In particular, $x \ll y$ (read, x directly precedes y) [or $x \gg y$ (read, x directly succeeds y)] if $x < y$ [$x > y$] and there exists no operation z such that $x < z < y$ [$x > z > y$].

Definition 2.2. The partial ordering between tasks given by the binary relationship $<$ is called precedence.

Definition 2.3. Given two tasks x and y, $x \sim y$ if x and y may be executed independently. That is, $x \neq y$, $x \not< y$, and $x \not> y$.

Definition 2.4. (Tree Precedence) The precedence of all tasks is called a tree if

1. There exists one and only one task x such that for all tasks y, $x \neq y$, $y < x$. The task x is called the root task (x is the last task).

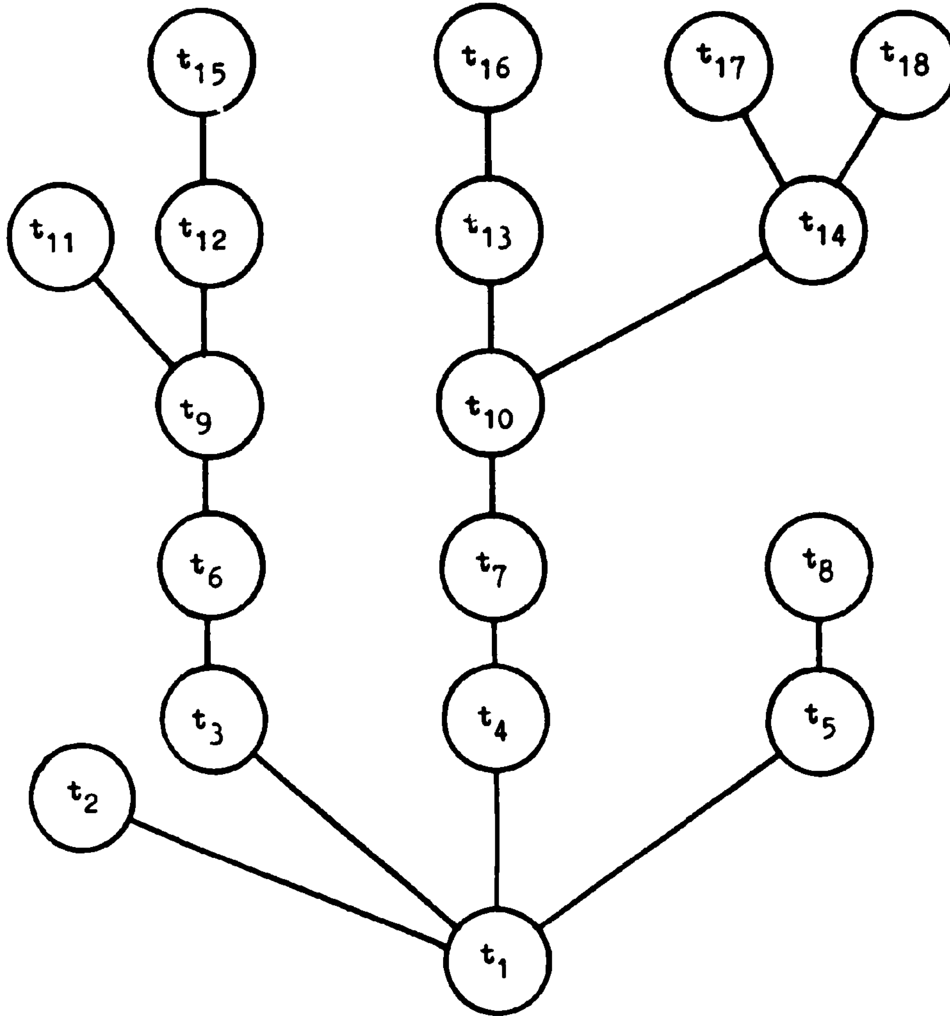
and 2. For each task y where y is not the root task, there exists one and only one task z such that $y \ll z$. (Each task except the last task has one and only one successor.)

Hu [1961, cf. Hsu 1966] developed the solution to a special $m \times n$ sequencing subproblem. For ten years, his algorithm remained the only major contribution to this area. The Hu problem is described in Problem 2.2. Figure 2.3 shows an example of the tasks. Algorithm 2.1 states Hu's "cutting the longest queue" procedure for two processors ($m = 2$).

Problem 2.2. Hu's Problem

Find the optimal schedule for the $m \times n$ sequencing problem in which all tasks have 1-unit length and tree precedence.

18 1-unit tasks with tree precedence



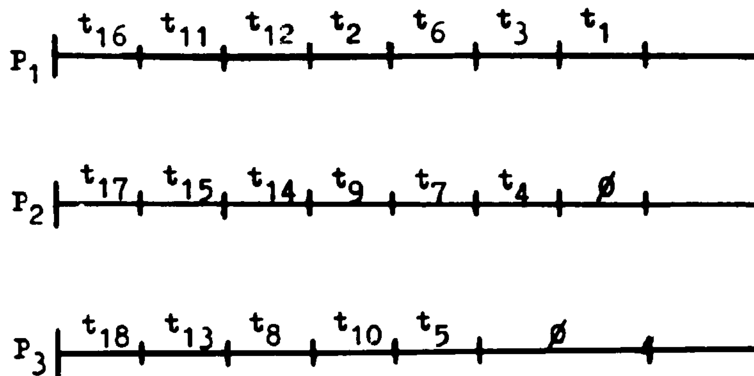


Figure 2.3. **Hu's "cutting the longest queue" algorithm**

Algorithm 2.1. Hu's solution to Problem 2.2 for $m = 2$

1. For each task in the tree precedence, calculate its distance from the root task.
2. At each instant that a new task is sought by a processor, assign the task farthest from the root task with all preceding tasks completed. Ties are broken at random. [Hu 1961]

Figure 2.4 shows that Hu's algorithm is not extendable to tasks of arbitrary length with tree precedence.

Before continuing the discussion of related results, a definition of a more complex kind of precedence is needed.

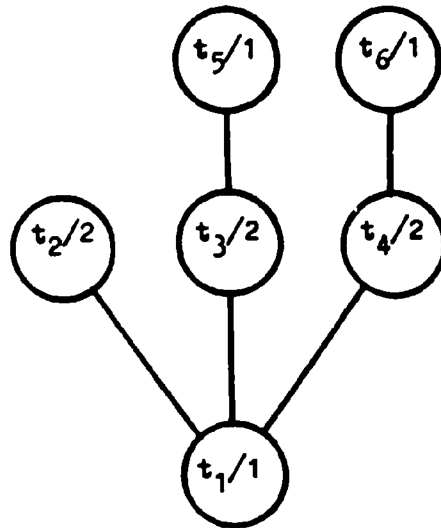
Definition 2.5. The precedence of all tasks is called acyclic if for all tasks, possibly with a partial ordering $<$, there is no task x such that $x < x$.

Chandy, Dickson, and Ramamoorthy [1972a, 1972b] observed that at least one of the many solutions which Hu's algorithm yields by breaking ties in different ways is indeed an optimal solution when all tasks are of 1-unit length with acyclic precedence and there are two processors ($m = 2$). They call the algorithm the Highest Level First (HLF) algorithm. Its significance, however, is diminished by a lack of a decision rule for determining efficiently the optimal solution from this still large set of possible solutions. Also we find Chandy's result to be a corollary of the concurrent work by Coffman and Graham [1972] which is described below.

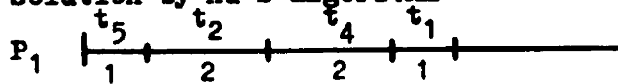
2.5. One Unit Tasks with Acyclic Precedence

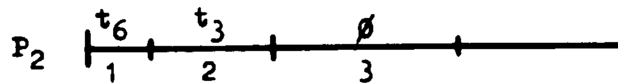
An algorithm by Coffman and Graham is most relevant to the work presented here. Their result is an algorithm which is at once effective and efficient. They have limited their scope to Problem 2.3.

6 1-unit and 2-unit tasks with tree precedence

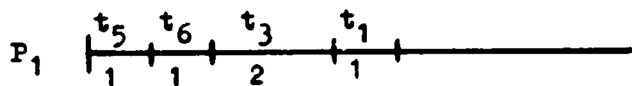


Solution by Hu's algorithm





Optimal schedule



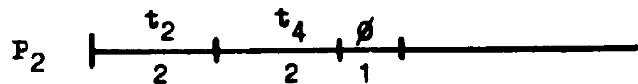


Figure 2.4. Hu's algorithm for unequal task lengths

Problem 2.3.

Find the optimal schedule for the $2 \times n$ sequencing problem in which all tasks have 1-unit length and acyclic precedence.

Figure 2.5 shows an example of the tasks in Problem 2.3 and a schedule from the Coffman-Graham algorithm. The strategy of the algorithm is straightforward. Initially, a task with no successors is assigned the label 1. Then after k-1 tasks are assigned labels 1, 2,...,k-1, a task is labelled k if

1. all its successors have received labels, and
2. the set of decreasing integer labels of the immediate successors of x is less than or equal to the set of decreasing integer labels of the immediate successors of all other tasks. (Ties are broken arbitrarily.)

The schedule is formed by selecting at each instant the task with the largest label with all predecessors completed.

If we denote the list produced by the Coffman-Graham algorithm by L^* and the length of time to complete the schedule generated with list L by $\omega(L)$, Theorem 2.6 from Coffman and Graham holds.

Theorem 2.6. For a set of one unit tasks with acyclic precedence, $\omega(L^*) \le \omega(L)$ for all lists L. [Coffman and Graham 1972]

2.6. Two Observations

The ideas reviewed in the previous section are the initial results needed to solve efficiently the general $m \times n$ sequencing problems with an arbitrary number of processors and an arbitrary number of tasks. The methods of solution are varied. Yet the resulting algorithms are efficient. Efficient extensions to these ideas will be another step in the direction of a complete solution to the $m \times n$ sequencing problem.

19 1-unit tasks with acyclic precedence

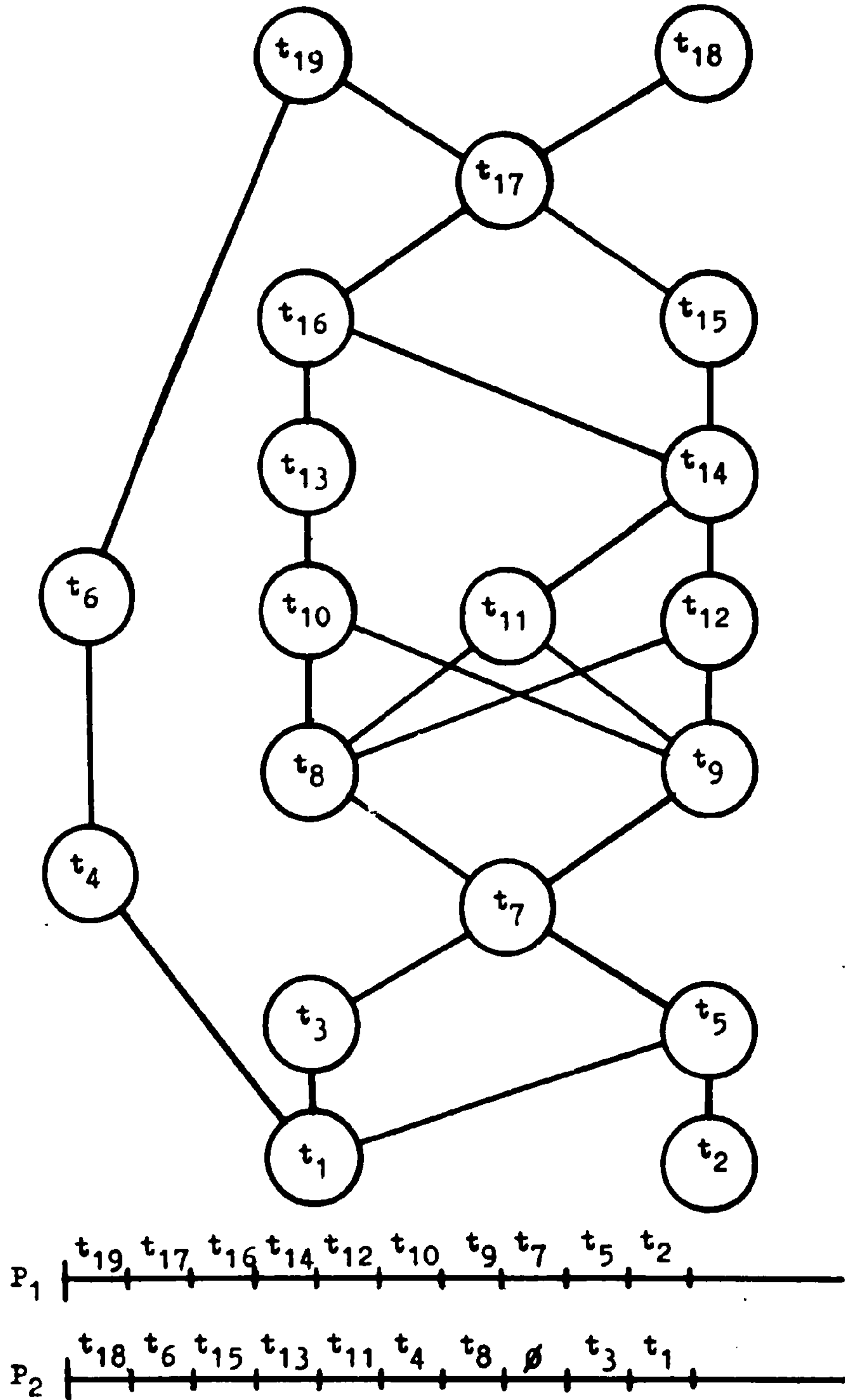


Figure 2.5. Coffman and Graham algorithm example [1972]

In addition, researchers are attempting to show that some sub-problems of the $m \times n$ sequencing problem have only solutions which grow exponentially with the size of the problem. There are no published results from this research. But such work in the inherent complexity of a problem may prove useful in bounding the algorithmic complexity limits which the sequence problem researcher may expect.

Chapter 3

One-Unit and Two-Unit Tasks

3.1. The Problem Statement

The work of Hu [1961] and Coffman and Graham [1972] provide solutions to problems composed entirely of 1-unit tasks. In our work we define a problem composed instead of 1-unit tasks and 2-unit tasks. Our problem, like Coffman and Graham's, involves only two processors ($m = 2$). Problem 3.1 and Problem 3.2 define the problems solved here. Figure 3.1 and Figure 3.2 show examples of these problems.

<u>Problem 3.1</u>. Tree Precedence Problem

Find an optimal schedule on two processors ($m = 2$) for a set G of n tasks with 1-unit and 2-unit lengths and tree precedence.

Before stating Problem 3.2 we define several concepts concerning the precedence of the tasks. These ideas are demonstrated in Figure 3.2.

<u>Definition 3.1</u>. A task x in a set of tasks G is called an <u>initial task</u> if there exists no task y such that $y \prec x$.

<u>Definition 3.2</u>. A task x in a set of tasks G is called a <u>terminal task</u> if there exists no task y such that $x \prec y$.

<u>Definition 3.3</u>. A set or sets of tasks with tree precedence, A, is <u>maximally connected</u> to another set or sets of tasks with tree precedence, B, if each terminal task of A is a predecessor of each initial task of B.

<u>Definition 3.4</u>. A set of tasks G with acyclic precedence has <u>p tree-restricted acyclic precedence</u> if G consists of p sets of tasks with tree precedence, $A_1, A_2, \ldots, A_p$, such that A_{i-1} is maximally connected to

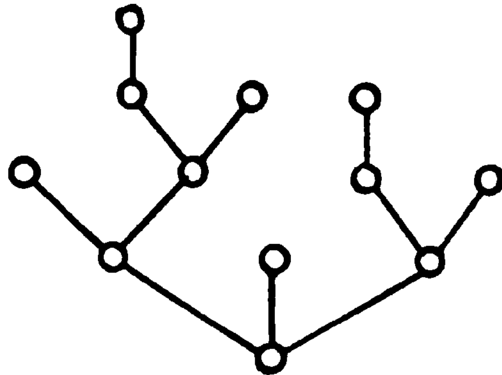


Figure 3.1. Example of set G in Problem 3.1 (tree precedence)

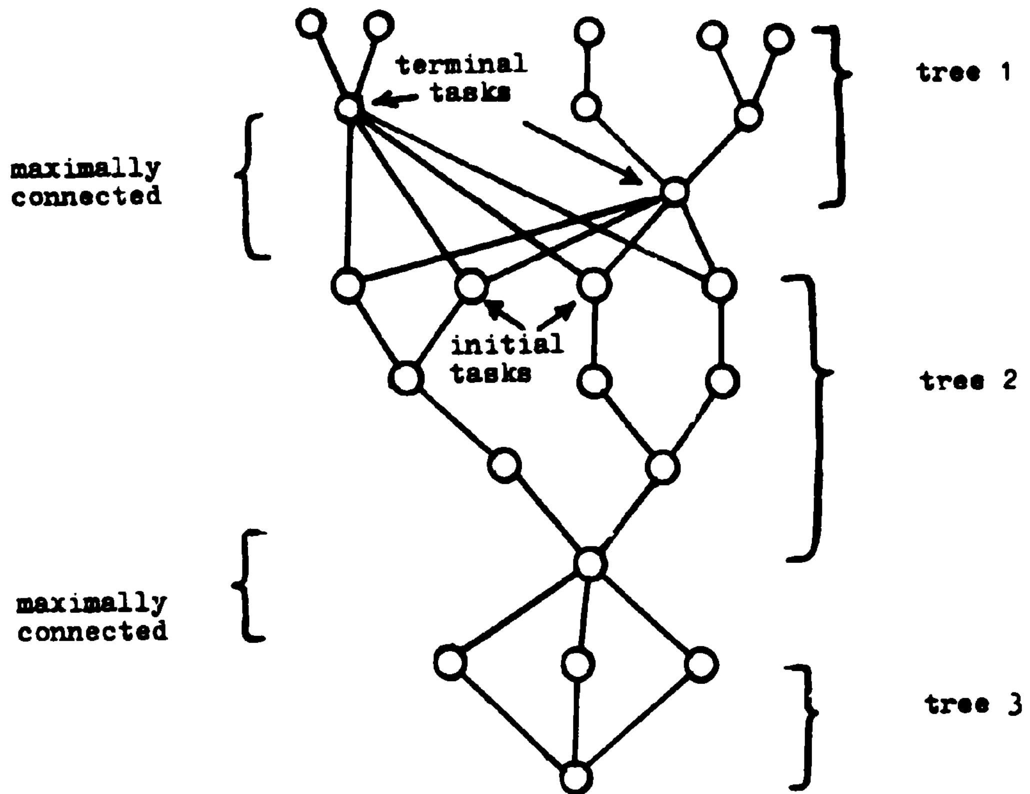


Figure 3.2. Example of set G in Problem 3.2 (3 tree-restricted acyclic precedence)

A_i, $i = 2, 3, \ldots, p$.

Problem 3.2. The Tree-restricted Acyclic Precedence Problem

Find an optimal two processor schedule for a set G of n tasks with 1-unit and 2-unit lengths and p tree-restricted acyclic precedence.

The algorithm to solve the tree precedence problem is developed in two steps. We do this both for simplification and for clarification of the algorithmic strategy. The algorithm for the solution of the tree-restricted acyclic precedence problem is closely related to that for the tree precedence problem. Before introducing the intermediate problem in Problem 3.3 we require two definitions about precedence.

Definition 3.5. A chain is a set of tasks $t_1, t_2, \ldots, t_r$ such that

$t_1 \ll t_2 \ll \ldots \ll t_r$.

Definition 3.6. The tasks of G are said to have chain precedence if each task in G is a member of one and only one chain (not necessarily the same chain).

Problem 3.3. The Chain Precedence Problem

Find an optimal two processor schedule for a set G of n 1-unit and 2-unit tasks with chain precedence.

Figure 3.3 shows an example of the chain precedence problem.

We solve the tree precedence problem and the tree-restricted acyclic precedence problem. Algorithm 3.1 is a task labeling procedure which is used by the succeeding algorithms. Algorithm 3.2, Algorithm 3.3 and Algorithm 3.4 treat the chain precedence problem, the intermediate problem. First, Algorithm 3.2 is a procedure for scheduling the tasks with the same label. Algorithm 3.3 then combines the schedule produced by Algorithm 3.2. However, the solution is not necessarily

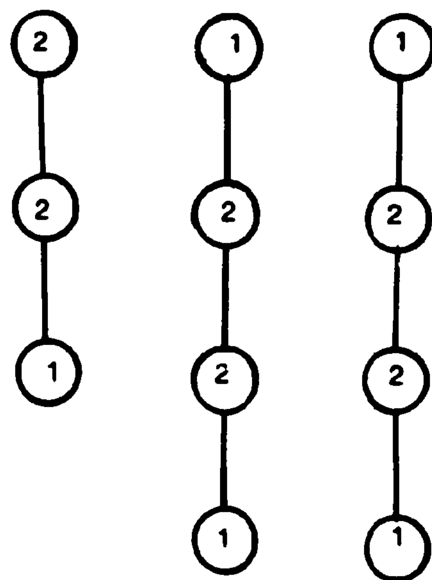


Figure 3.3. Example of set G in the chain precedence problem

optimal. The cases that lead to nonoptimal solutions are treated by Algorithm 3.4. Algorithm 3.3 and Algorithm 3.4 then combine with Algorithm 3.5 to solve the tree precedence problem. Finally, Algorithm 3.6 solves the tree-restricted acyclic precedence problem.

3.2. Development of the Solution to the Chain Precedence Problem

The situation posed in the chain precedence problem concerns only chains of tasks and is a subproblem of the tree precedence problem. Figure 3.4 is an example of how a set of chains may be modified to become a tree of tasks with one root task. A nonexistent root task is added which has as its predecessor all terminal tasks of the chains.

The development of the algorithm for the chain precedence problem is a series of three basic algorithms which lead to a possibly nonoptimal solution. A fourth algorithm modifies the nonoptimal solution to an optimal solution. Since the chain precedence problem is a subproblem of the tree precedence problem, the proof of the solution to the chain precedence problem is omitted. Only the proof of the solution to the tree precedence problem is stated.

3.2.1. Algorithm 3.1--Labelling

Algorithm 3.1 is a procedure for accomplishing the labelling of the tasks. This algorithm is applicable to tasks with tree precedence as well as chain precedence and is similar to Hu's algorithm. The algorithm begins by labelling each initial task with the label 1. All other tasks receive an integer label one greater than the largest label of its predecessors.

<u>Algorithm</u> <u>3.1</u>. Labelling

1. Label each initial task with level number 1.
2. For each task for which each of its immediate predecessors has

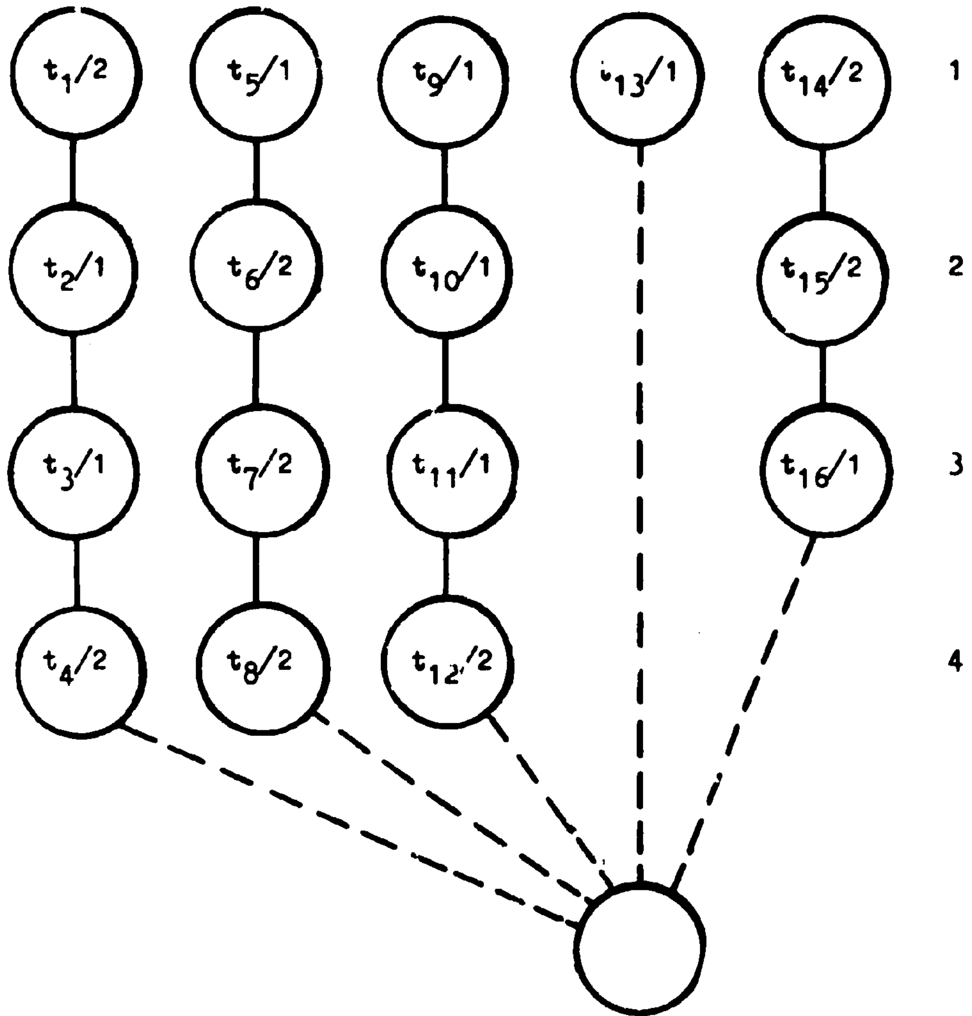


Figure 3.4. Chains and trees

been assigned level numbers, label the task with the maximum level number of its predecessors plus one.

3. Repeat Step 2 until each task has been assigned a level number.

Figure 3.5 shows an example of the labels produced by Algorithm 3.1. Throughout this chapter the largest level number assigned by the labelling procedure is called M.

In an analysis of Algorithm 3.1 each task must be visited once for labelling. However, when labelling a specific task x all immediate predecessors of x must be examined. Since each task in a tree is an immediate predecessor of at most one task, each task except the terminal tasks must be visited only twice. Therefore, for n tasks approximately 2n operations are required. The computation for Algorithm 3.1 is, therefore, of order n, $O(n)$.

Having found level numbers for each task by Algorithm 3.1 we then may refer to the number of tasks at a given level or merely to the level number itself. The following definitions are helpful.

<u>Definition 3.7</u>. For each task x in G, L(x) is the <u>level number</u> or <u>label</u> of task x.

<u>Definition 3.8</u>. For level k in G, <u>N(k)</u> is the number of tasks at level k remaining to be scheduled.

Using Figure 3.6, the concepts of Definition 3.7 and 3.8 are clear. For example, $L(t_5) = 2$ and $N(3) = 2$.

Corollary 3.1 is immediately apparent from the labelling algorithm and Definition 3.7. The corollary states that the predecessors of a task have smaller level numbers.

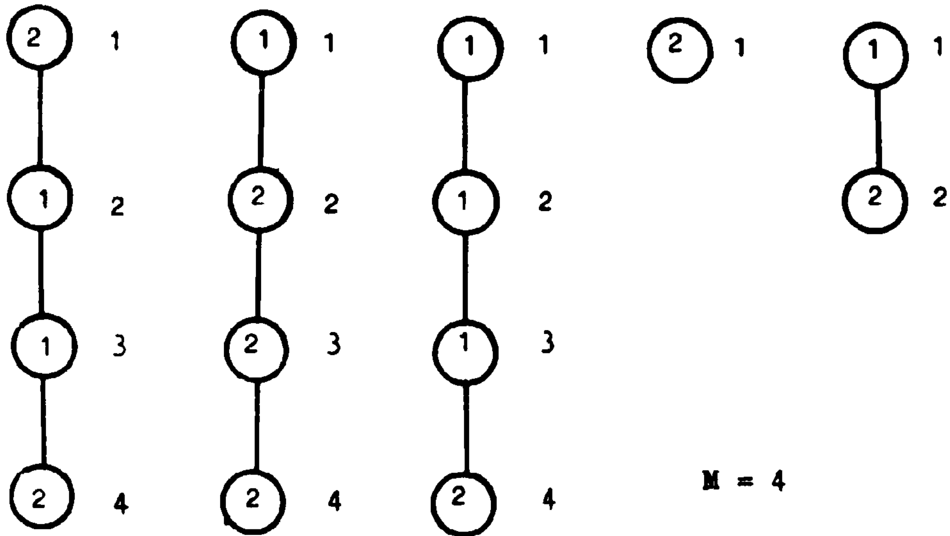


Figure 3.5. Example of the labelling algorithm

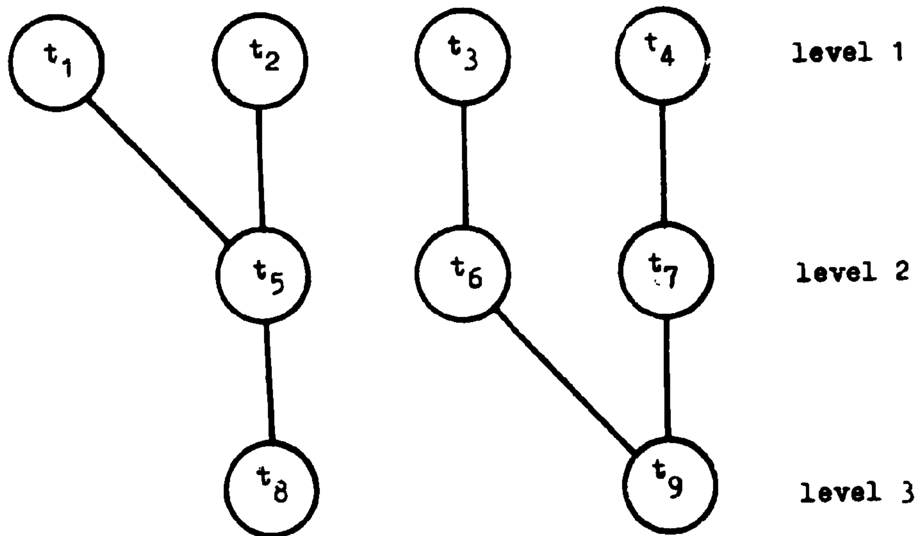


Figure 3.6. Level notation

Corollary 3.1. For tasks x, y in G and $x \prec y$, $L(x) < L(y)$.

The value of $N(k)$ for each level k in G changes as a scheduling algorithm progresses. For example, if level j has six tasks initially and one task is assigned before the remaining five tasks, $N(j) = 5$ after the single task is assigned.

3.2.2. Algorithm 3.2--Individual Level Scheduling

The second algorithm, Algorithm 3.2, schedules tasks with the same level number without consideration of the tasks from other levels. We then characterize the individual schedules to observe their form before incorporating them into a complete schedule in Algorithm 3.3.

In Algorithm 3.2 all 2-unit tasks are assigned before 1-unit tasks. A task is assigned when a processor calls for a new task. Figure 3.7 shows an example of the use of the algorithm.

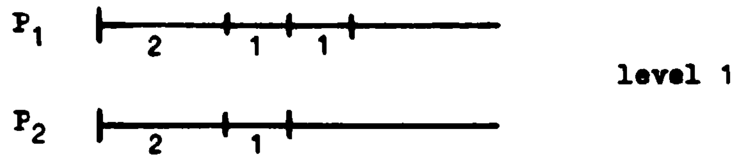
Algorithm 3.2. Individual Level Scheduling

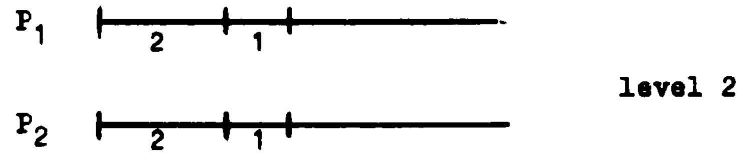
Let the two processors be P_1 and P_2. For a given level,

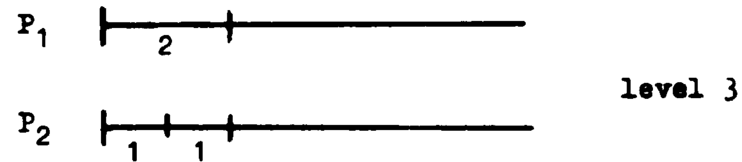
1. Order the tasks into a list L so that 2-unit tasks precede 1-unit tasks.
2. When a processor needs a task, assign the next unassigned task in list L. If both processors need a task simultaneously, assign the next unassigned task in list L to P_1 and the second unassigned task in list L to P_2.

Algorithm 3.2 schedules the $N(K)$ tasks of each level k. The ordering of Step 1 is a simple procedure; all 2-unit tasks must precede 1-unit tasks. The ordering is accomplished by creating a double-ended queue, or deque, in which one end is for 2-unit tasks and one end is for 1-unit tasks.

Two Processors







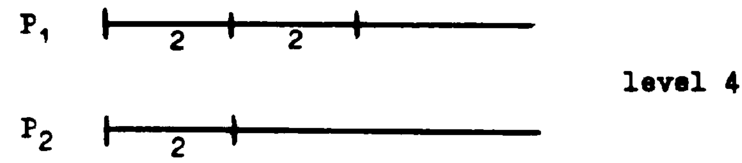


Figure 3.7. Use of Algorithm 3.2 in example of Figure 3.4

Each task is visited once during Step 1 for N(K) operations. In Step 2 each task is again visited to create the level's schedule; again there are N(K) operations. The total number of operations for a given level k is 2N(K) Algorithm 3.2 is then also of order n, O(n).

Algorithm 3.2 treats only 1-unit and 2-unit tasks on two processors. The algorithm is applicable to two processors which begin simultaneously as well as to processors which do not. With the condition that execution on processors P_1 and P_2 begins simultaneously several observations about the individual level schedules become apparent.

Corollary 3.2. If processors P_1 and P_2 begin execution simultaneously, Algorithm 3.2 produces a schedule in which processor P_2 completes all tasks either 0, 1, or 2 units before processor P_1.

Corollary 3.3. If processors P_1 and P_2 begin execution simultaneously, Algorithm 3.2 produces a schedule in which at most one task is executed on processor P_1 when P_2 becomes idle.

With the condition that processor P_2 begins execution before processor P_1 we make two additional observations. In particular, we are concerned with the cases where processor P_2 begins execution one unit or two units before processor P_1.

Corollary 3.4. If processor P_2 begins execution 1 or 2 units before processor P_1, Algorithm 3.2 produces a schedule in which one processor completes all tasks 0, 1, or 2 units before the other processor.

Corollary 3.5. If processor P_2 begins execution two units before processor P_1, Algorithm 3.2 produces a schedule in which at most two tasks begin execution on processor P_2 before processor P_1 begins execution.

3.2.3 Solution to the Chain Precedence Problem

Using the labeling procedure of Algorithm 3.1 and the schedules of the individual levels produced by Algorithm 3.2, Algorithm 3.3 produces a schedule for the tasks in the chain precedence problem. However, Algorithm 3.3 does not always produce an optimal schedule. The discussion following the algorithm points out the failures. Algorithm 3.4 then summarizes the modifications required in Algorithm 3.3. But in order to make Algorithm 3.4 applicable to the tree precedence problem also, new definitions are needed. Algorithm 3.3 and Algorithm 3.4 together produce an optimal solution to the chain precedence problem.

Algorithm 3.3 has three distinct sections. Steps 1, 2, and 3 use Algorithm 3.1 and Algorithm 3.2 to provide the initial labelling of the tasks and the schedules of individual levels. Scheduling begins with the terminal tasks and progress back to the initial tasks. Step 4 is used when the level to be scheduled and all the levels of predecessors have at least three tasks. The construction of the complete schedule from partial schedules of Algorithm 3.2 is summarized by Table 3.1 following Algorithm 3.3.

Steps 5, 6, 7, and 8 treat the case when the level to be scheduled has less then 3 tasks. Then no level scheduled earlier has had 3 or more tasks. Step 5 schedules all remaining tasks when only a single chain of tasks remains to be assigned. Step 6 schedules tasks when the level to be scheduled has less than three tasks. The processors are dedicated to the longest and second longest chains, respectively. These chains, exclusively, are assigned to these processors unless other chains equal the shorter chain in number of levels and the length of the queue on the processor is two or more units less than the other queue. Then a chain other than the dedicated chain may be scheduled. Essentially, the queues are maintained nearly equal in length until the level to be scheduled has more than two

tasks or all tasks are scheduled. Steps 7 and 8 detect the termination conditions and provide the iterative structure of the algorithm.

Algorithm 3.3. Basic Scheduling Algorithm for Problem 3.3

1. Using Algorithm 3.1 assign a level number to each task.
2. Schedule level M using Algorithm 3.2 with both processors beginning simultaneously.
3. Set the current level to level M-1.
4. If N(level) ≥ 3,
   - 4A. For level = level, level - 1, ... , 1:
     - 4A.1. Note the number of units U a processor is idle in the current schedule while the other processor executes some tasks.
     - 4A.2. Schedule the current level using Algorithm 3.2 with processor P_2 beginning execution U units before processor P_1.
     - 4A.3. Rearrange the tasks in the schedule of the current level so that no task performed on processor P_2 in the first U units is a successor of the last task(s) performed in the current schedule.
     - 4A.4. Combine the current schedule and the schedule of the current level.
   - 4B. The schedule is complete. Stop.
5. If N(k) = 1 for k = level, level-1, ... , 1, assign all tasks in order. Schedule is complete. Stop.
6. If N(level) < 3,
   - 6A. Assign a queue to the longest chain and the second queue to the next longest chain. Call these dedicated queues. Break

ties arbitrarily.

6B. Assign current level tasks to the preassigned queues.

6C. Define DQ_i, the deficiency of queue i, as the number of units queue i lags the other queue.

6D. If $DQ_i < 3$ and $N(level-1) \geq 3$, go to Step 8.

6E. If $DQ_i = 0$, go to Step 8.

6F. Let next level be greatest level number of the tasks available to be assigned but not dedicated to the longer queue.

6F.1. If a task in next level is from the chain dedicated to queue i,

6F.1A. If the task's length is less than or equal to DQ_i, assign the task to queue i and repeat Step 6F.

6F.1B. Otherwise, go to Step 8.

6F.2. If a task in next level is not from the chain dedicated to queue i,

6F.2A. If $DQ_i \geq 2$, and if by assigning the task DQ_i remains $DQ_i \leq 1$, then assign the task and repeat Step 6F.

6F.2B. Otherwise, go to Step 8.

7. If $N(level-1) = 0$ or $N(k) = 1$ for $k = level-1, level-2, \ldots, 1$, assign all tasks to the dedicated queue. Schedule is complete. Stop.

8. Let level equal level - 1. If $N(level) \geq 3$, go to Step 4. Otherwise go to Step 6B.

In the schedules produced by Algorithm 3.2 and in some of the schedules produced by Algorithm 3.3 no processor is idle while the other processor is executing except at the end of the assignment. In these compacted schedules we are interested in three possible forms of the assignment or partial assignment. Figure 3.8 shows examples of Form A, Form B, and Form C described in Definition 3.9.

Definition 3.9. An assignment or partial assignment in which both processors begin simultaneously and in which one processor completes execution two units, one unit, or zero units before the second processor is said to be of Form A, Form B, or Form C, respectively.

The forms described in Definition 3.9 are convenient for summarizing the assignments made in Step 4A of Algorithm 3.3. Table 3.1 provides this summary which is easily verified by case analysis. Here, X is the partial assignment called current schedule. Y is the assignment of the current level as found in Algorithm 3.2 with both processors' beginning execution simultaneously. Later we refer to the operation of Table 3.1 as the · operation.

X \ Y	Form A	Form B	Form C
Form A	C	B	A or C#
Form B	B	C	B
Form C	A	B	C

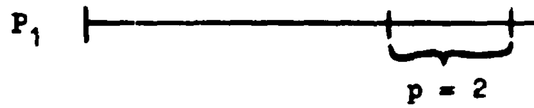
Table 3.1. Assignment of Forms

(Form A)·(Form C) = Form A if Form C contains no 1-unit tasks or only contains three tasks with the 2-unit task a predecessor of the last task in Form A. In all other cases (Form A)·(Form C) = Form C.

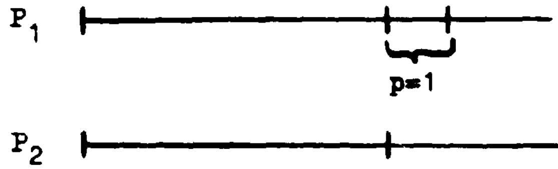
One observation is immediately apparent about a complete assignment of 1-unit and 2-unit tasks which has no idle time on either processor except possibly at the end of the assignment.

Corollary 3.6. A complete assignment of 1-unit and 2-unit tasks which is of Form B or Form C is optimal.

Form A:



Form B:



Form C:

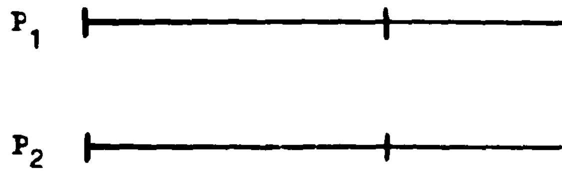


Figure 3.8. Form A, Form B, and Form C

A complete assignment which is of Form A may or may not be optimal. Corollary 3.7 describes a case that is clearly optimal since only 2-unit tasks are involved. The discussion following the corollary outlines the shortcomings of Algorithm 3.3 and the means for their elimination.

Corollary 3.7. A complete assignment of 2-unit tasks which is of Form A or Form C is optimal.

In the analysis of Algorithm 3.3, we first focus on Step 4. Indeed, if $N(k) \geq 3$, no succeeding steps of the algorithm are used. For convenience in understanding the affect of Step 4, we refer to Table 3.1 throughout the discussion. Using the terminology of this table we note that it is possible for an assignment to end in Form A and not be optimal. Figure 3.9 shows four situations in which a nonoptimal Form A assignment occurs. Later we show that only these four kinds of situations occur if a level k with $N(k) \geq 3$ contains 1-unit tasks. The shortcomings of Algorithm 3.3 follow:

I. The first situation, depicted in Figure 3.9(A), is such that the last level assigned with 1-unit tasks (level 2) is of Form B by Algorithm 3.2. The partial assignment after level 3 is assigned also is of Form B. Instead of the Form C obtained by the · operation after the assignment of level 2 we desire to obtain Form A. To accomplish this result in general where level i has the last 1-unit task, the last 1-unit task in level i should be moved to the other processor. If a conflict with level i+1 occurs, it must be resolved by rearrangement of tasks. Also although one processor executes for two units while the other is idle, no conflict occurs at level i-1. Since $N(i-1) \geq 3$ and level i-1 has all 2-unit tasks, at least one task of level i-1 may be found to

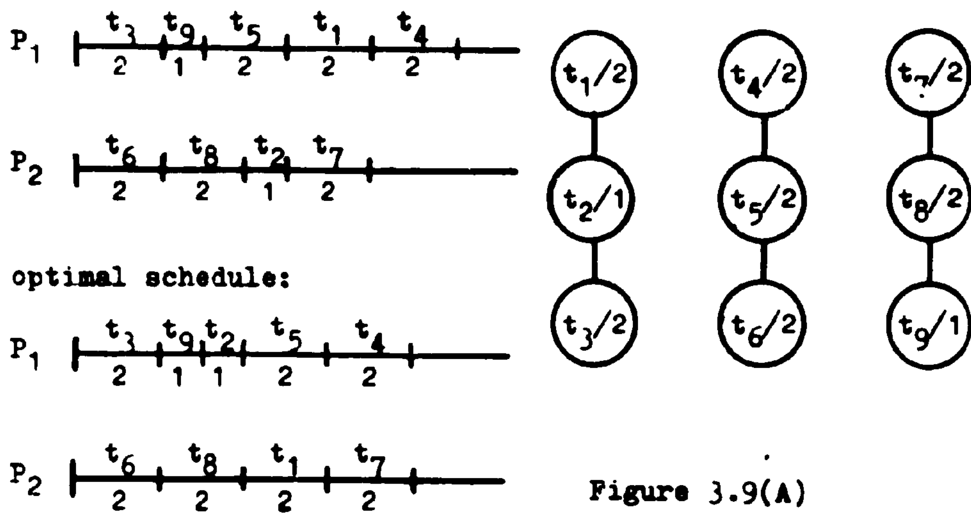


Figure 3.9(A)

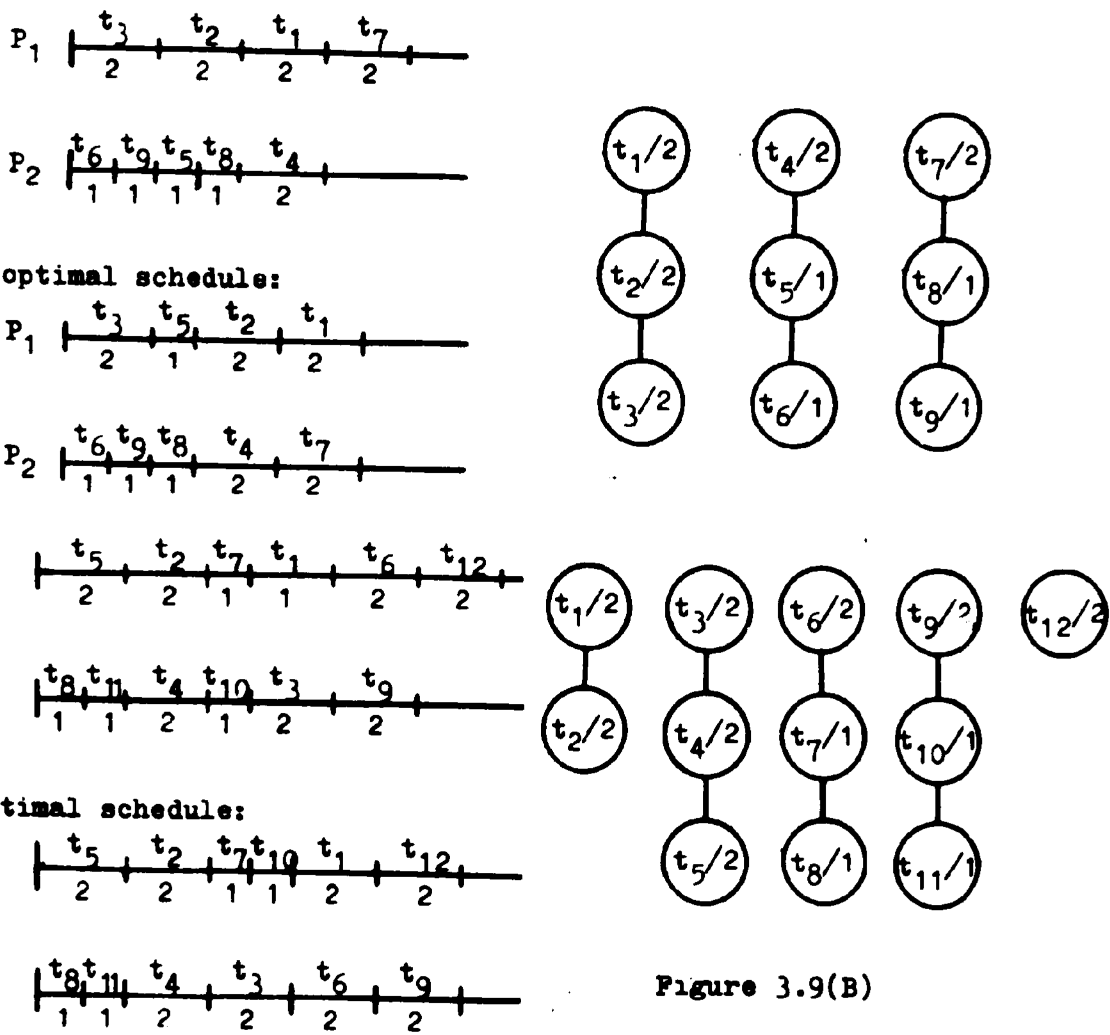


Figure 3.9(B)

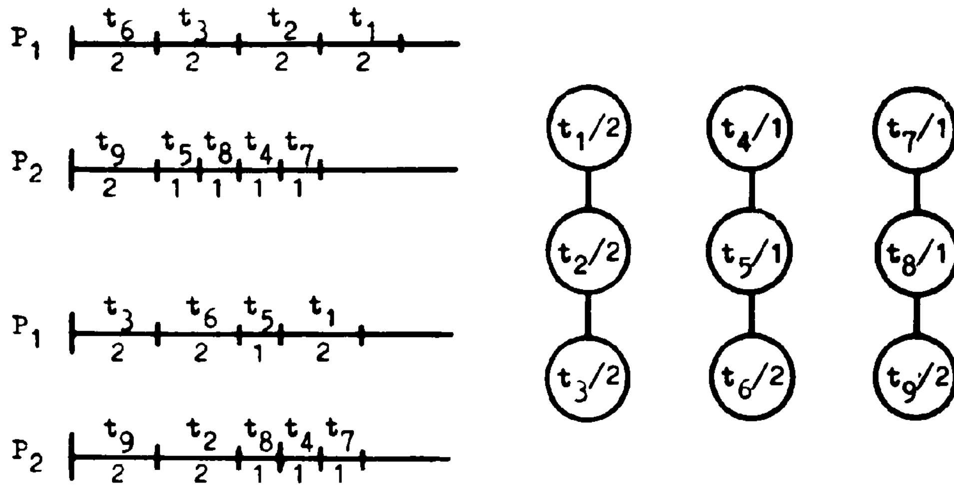


Figure 3.9(C)

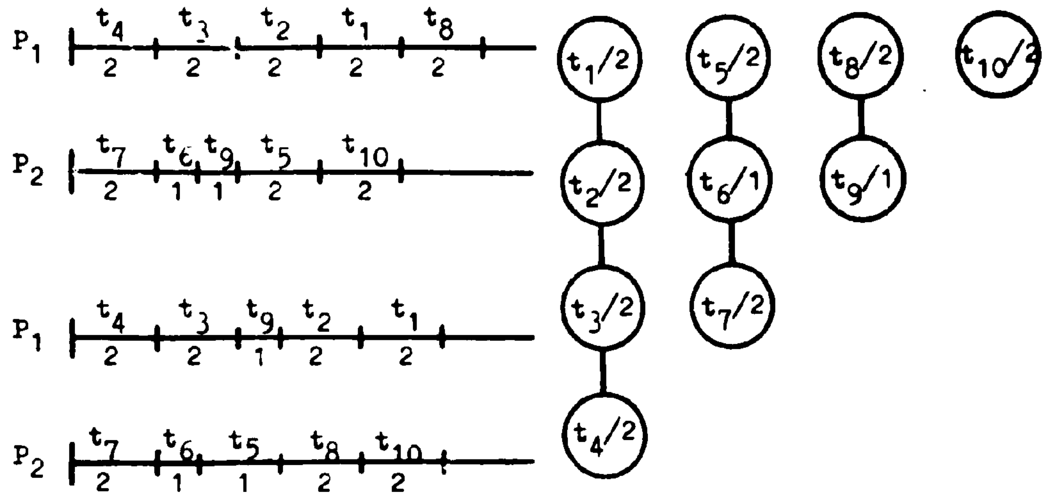


Figure 3.9(D)

Figure 3.9. Modifications

execute during the two units even if the two units represent two tasks.

II. The second situation, depicted in Figure 3.9(B), is such that the last level with 1-unit tasks in level i ($i = 2$ in both cases of Figure 3.9(B)) is of Form C. Then either $N(i) > 3$ and the partial assignment through level i+1 is of Form A or Form C, or $N(i) = 3$ and the partial assignment through level i+1 is of Form C. In both cases, the remedy is the same as that described in I above.

III. In the third situation, shown in Figure 3.9(C), level 1 is of Form C and $N(1) = 3$ and the partial assignment through level 2 is of Form A. Level 1 is the last level with 1-unit tasks. In general the last level with 1-unit tasks is of Form C with $N(i) = 3$ and the partial assignment through level i+1 is of Form A. We then want to back up and find a level j, $j \geq i$, of Form C and $N(j) = 3$. In the figure, $j = 2$. If $N(j+1) \geq 3$, one of the three tasks in level j+1 does not conflict with the 2-unit task of level j. This nonconflicting task should be assigned last in level j. Level j through level i are assigned by the usual operation. In Figure 3.9(C), then, level 3 is rearranged so that task t_6 is assigned last. Levels 2 and 1 are assigned in the usual way.

IV. In some cases no level j may be found such that $N(j+1) \geq 3$ and $N(j) = 3$. This situation occurs in the example shown in Figure 3.9(D). In the figure level 2 is of Form C, $N(2) = 3$, $N(1) = 4$ and the partial assignment through level 2 is of Form A. In general, some level i is of Form C, $N(i) = 3$, $N(i-1) \geq 3$, and the partial assignment through level i is of Form A. One of the 1-unit tasks of level i is placed before the 2-unit task of level i. One of the 2-unit tasks of level i-1 does not conflict and may be assigned following the second 1-unit task of level i followed itself by the predecessor of the first 1-unit task of level i. The remainder of the assignment of level i proceeds as in Algorithm 3.2

If no 1-unit tasks exist in a level k such that $N(k) \geq 3$, it is possible that 1-unit tasks exist in some other level j where $N(j) < 3$. Again we may transform the schedule. One queue will be longer than the other. We use here and prove later in relation to Algorithm 3.5 that all tasks in the longer queue are related (Lemma 3.1). Figure 3.10 shows two typical cases which need transformation.

V. In Figure 3.10 a 1-unit task occurs in the longer queue (P_1) or in the shorter queue (P_2). Find the last 1-unit task assigned and call it task D. When task D was assigned, a task F either started execution on the other processor or was midway through execution. Either D or F was on the shorter queue, and a task E was available for assignment in its place. If tasks D and F begin execution simultaneously, place D in the other queue before F. Otherwise, place D on the other queue after F. Replace D with task E.

E may have been assigned before level k where $N(k) \geq 3$ and assigned to the shorter queue. In this case replace it with E', which is either a task from level k or a predecessor of T. Repeat this process until level k is assigned. Assign level k so that no conflict occurs.

In the case when E is from level j, j = k, k-1, ... , 1, replace E with a task from level k. Assign level k so that no conflict occurs.

As is shown in Theorem 3.1, level k may be assigned since $N(k) \geq 3$. The structure of the partial assignment is changed either from Form A to Form C or from Form C to Form A.

These corrections are performed by Algorithm 3.4. Algorithm 3.3 and Algorithm 3.4 produce the solution to the chain precedence problem. This fact is stated without proof since this problem is a special case of the tree precedence problem.

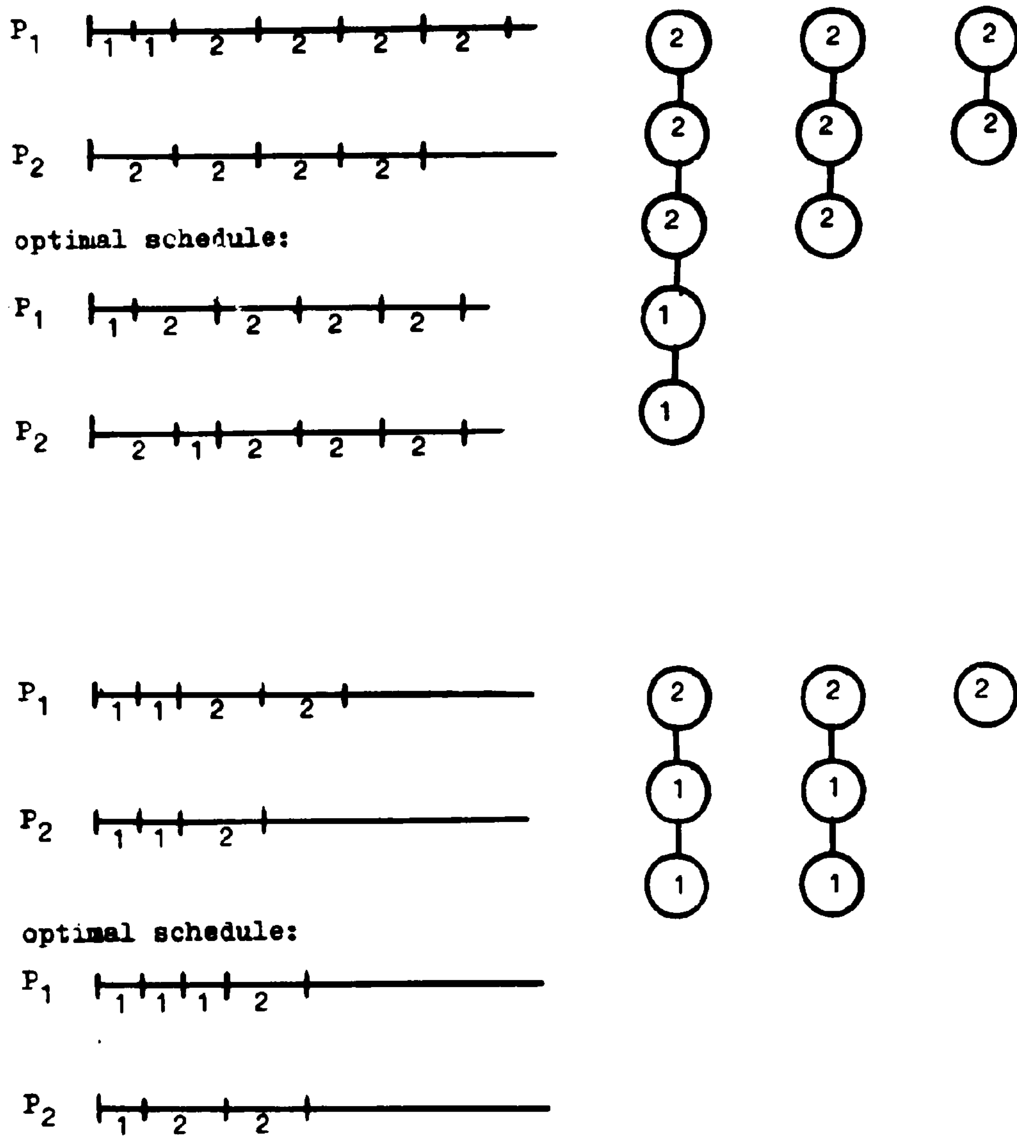


Figure 3.10. 1-unit tasks in Algorithm 3.3

Before stating Algorithm 3.4 for chain precedence, we extend the modifications slightly for tree precedence. This extension avoids the repetition of the algorithm later. To accomplish this we present three definitions. Figure 3.11 shows examples of the situations of Definitions 3.11, 3.12, and 3.13.

Definition 3.10. The longer queue is the queue dedicated to the subtree with the highest level number or simply the queue with more time units assigned.

Definition 3.11. An assignment break occurs at time T if

1. the longer queue completes execution of a task x at time t,

and 2. insufficient tasks y, $y \sim x$, remain to be assigned to the second queue to extend beyond time T.

Definition 3.12. An assignment stop occurs at time T if

1. the longer queue completes execution of a task x at time T,

and 2. x is in level k such that $N(k) = 1$,

and 3. $N(k-1) = 2$ such that both tasks are predecessors of x.

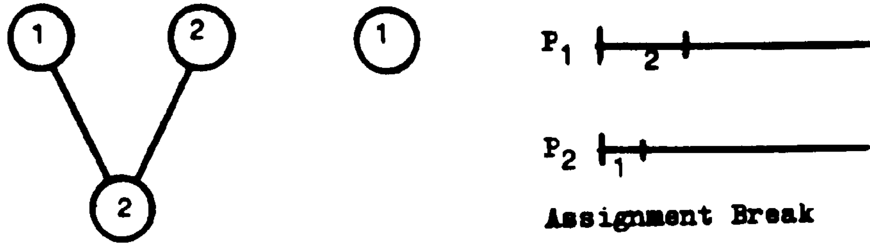
Definition 3.13. An assignment fork occurs at time T if

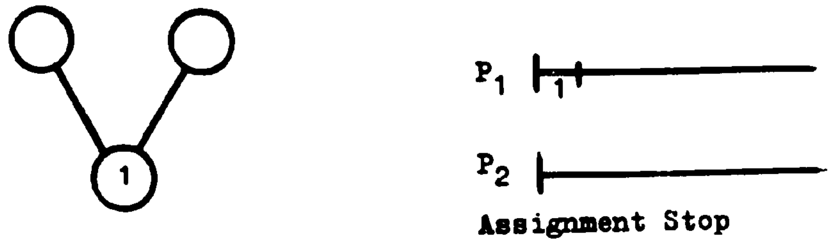
1. the longer queue completes execution of a task x at time T,

and 2. x is in level k such that $N(k) \leq 2$,

and 3. insufficient tasks y, $y \sim x$, remain to be assigned to a second processor to extend beyond time T+1,

and 4. an assignment break did not occur at time T or T+1.





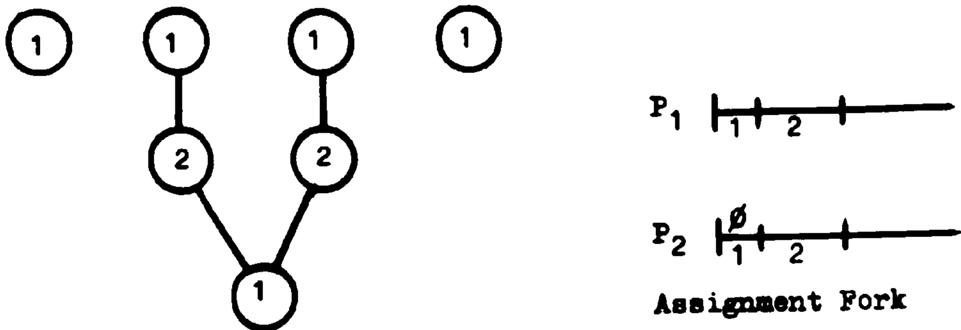


Figure 3.11. Examples of definitions

Algorithm 3.4 corrects the failures of Algorithm 3.3 but uses the terminology of tasks with tree precedence above. The algorithm is applicable to both the chain precedence problem and the tree precedence problem. The strategy of Algorithm 3.4 parallels the five points made in the discussion of Algorithm 3.3's shortcomings.

Algorithm 3.4.

1. If the schedule is not of Form A, or if the schedule does not contain 1-unit tasks, or if the schedule is equal in length to the longest chain, or if no 1-unit tasks appear after the last assignment fork or assignment break, Algorithm 3.4 does not apply. The current schedule is optimal.
2. Find the largest numbered level k such that $N(k) \geq 3$ when assigned in the current schedule.
3. If at least one 1-unit task occurs in levels k-1, k-2, ... , or 1, find the smallest value i such that level i contains a 1-unit task.

   3A. If level i is of Form B by Algorithm 3.2, then by Table 3.1 the partial assignment k, k-1, ... , i+1 is of Form B. Take the last 1-unit task assigned in level i, and assign it to the other processor.

   3B. If level i is of Form C by Algorithm 3.2 and if $N(i) > 3$, or if level i is of Form C by Algorithm 3.2 and if $N(i) = 3$ and if the partial assignment of levels k, k-1, ... , i+1 is of Form C, take the last 1-unit task assigned in level i and assign it to the other processor.

   3C. If level i is of Form C by Algorithm 3.2 and if $N(i) = 3$

and if the partial assignment of levels k, k-1, ... , i+1 is of Form A, find the largest number j such that level j is of Form C by Algorithm 3.2, $N(j) = 3$ and $k > j \geq i$. If such a j exists, select one task from the last three tasks assigned in level j+1 which does not conflict with the 2-unit task of level j. Assign the nonconflicting task from level j+1 last. Assign levels j, j-1, ... , i by the · operation.

3D. If level i is of Form C by Algorithm 3.2, and if $N(i) = 3$, and if $N(i-1) \geq 3$, and if the partial assignment of levels k, k-1 ... , i is of Form A, place the 1-unit task of level i before the 2-unit task of level i on the longer queue. Place one of the nonconflicting tasks from level i-1 on the second queue followed by a task which does not conflict with the 2-unit task of level i.

In all cases complete the assignment by assigning levels i-1, i-2, ... , 1 by the · operation.

4. If no 1-unit tasks occur in levels k, k-1, ... , or 1 and a 1-unit task occurs in levels M, M-1, ... , or k+1 after the last assignment fork or assignment break,

4A. Find the last 1-unit task assigned and call it task D. When task D was assigned, a task F was either starting execution on the other processor or was midway through execution. Place D on the other queue to start before F if they had begun execution simultaneously or, otherwise, immediately after task F. Either task D or task F was not on the longer queue and another task E could have been assigned in its place. Replace task D with task E. If E

had already been assigned to the shorter queue, replace E with its predecessor. Continue this replacement until either no more predecessors of E have been assigned to the shorter queue before level i or a predecessor of E may be replaced by a task from level i.

A proof of the correctness of Algorithms 3.3 and 3.4 is required here. However, the solution of the chain precedence problem is merely a subcase of the solution to the tree precedence problem which follows. Therefore, we defer a proof of the algorithm until Algorithm 3.5 is stated.

3.3. Solution to the Tree Precedence Problem

In this section we extend the solution of the chain precedence problem to the tree precedence problem. Instead of considering only tasks with chain precedence, we permit tasks to be related with tree precedence. The resulting algorithms for the solution to the new problem is Algorithm 3.4 and Algorithm 3.5. Its form is very similar to that of Algorithm 3.3. However, the situations which involve assignment stops and assignment forks complicate the algorithm.

Steps 1 and 2 of Algorithm 3.5 use Algorithm 3.1 to label the tasks. Step 3, which corresponds to Step 4 of Algorithm 3.3, treats a level and all later levels having three or more tasks. All other steps treat the situations where a level with 3 or more tasks has not yet been located. Step 4 detects the case where only one subtree remains from the original tree. In this case the partial solution may be set aside, and the set of tasks in the subtree may be considered to be a new problem.

Steps 5 through 13 isolate and treat the special cases where the number of tasks in the level to be scheduled is less than three. We call

a subtree of unassigned tasks with the largest level number the longest subtree. Similarly, we call a subtree of unassigned tasks with the second largest level number the second longest subtree. Step 5 then assigns the two processors to the longest and second longest subtrees. Steps 6, 7, 8, and 9 then assign tasks from these two dedicated subtrees so that the processor queues remain within two units of each other. If other subtrees have the same length as the shorter dedicated subtree, Step 9 selects tasks from these subtrees. Step 7 detects the time when the number of tasks to be scheduled in the next level is three or greater. Then the queues are ready for the use of Step 3. Steps 10 and 11 detect the termination conditions for the algorithm or the existence of only one subtree. Step 12 detects an assignement stop within the schedule. The elimination of time gaps within the queue schedule is handled in Step 7A and 9B.1. Steps 12C and 13 provide the iteration mechanism until Step 3 is applicable.

Algorithm 3.5.

1. Using Algorithm 3.1 assign a level number to each task.
2. Let the current level be equal to M, the largest level number.
3. If $N(M) \geq 3$,
   - 3A. Schedule level M by Algorithm 3.2.
   - 3B. Set level = level - 1.
   - 3C. For level = level, level-1, ... , 1,
     - 3C.1. Note the number of units, U, a processor is idle in the current schedule while the second processor executes some tasks.
     - 3C.2. Schedule the current level using Algorithm 3.2 with processor P_2 beginning execution U units before

processor P_1.

3C.3. Rearrange tasks of the current level in the schedule of Step 3C.2 so that no task performed on processor P_2 in the first U units is a successor of the last tasks performed in the current schedule.

3C.4. Combine the current schedule and the schedule of the current level.

3D. Go to Algorithm 3.4.

4. If only one subtree exists, schedule a single task on longer queue. Set aside the partial solution and delete the assigned task from the tree. Begin Algorithm 3.5 at Step 2 with the revised set of tasks.

5. If $N(M) < 3$, assign a queue to the subtree with the largest level number and the second queue to the subtree with the next largest level number. Break ties arbitrarily. Call these queues dedicated queues.

6. If $N(\text{level}) < 3$, assign the current level tasks to the dedicated queues.

7. If $DQ_i \leq 2$ and $N(\text{level}-1) \geq 3$,

7A. If a previous stop gap remains unfilled (see Step 12B), match the longer queues together and fill the gap. If $N(\text{level}-1) < 3$, go to Step 8.

7B. Assign tasks to the deficient queue so that $DQ_i = 0$. If this is impossible, assign tasks so that $DQ_i = 1$. If neither is possible, an assignment break exists.

7C. If $N(\text{level}-1) \geq 3$, set level equal to level-1 and go to Step 3B.

7D. If N(level-1) < 3, assign level-1 to the dedicated queues. Set level equal to level-1.

7E. If N(level-1) < 3, assign level-1 to the dedicated queues and set level equal to level-1.

7F. If no subtree remains, the assignment is complete. Otherwise, set level equal to level-1 and go to Step 3B.

8. If $DQ_i = 0$, go to Step 12.

9. Let the next level be the greatest level number of the tasks available to be assigned but not dedicated to the longer queue.

9A. If a task in the next level is from the subtree dedicated to queue i,

9A.1. If its length is less than or equal to DQ_i, assign it to queue i. Repeat Step 9.

9A.2. Otherwise, go to Step 12.

9B. If the task in the next level is not from the tasks dedicated to queue i and if $DQ_i \geq 2$ and if assigning the task level leaves $DQ_i \geq 1$,

9B.1. If a previous unfilled stop gap exists (see Step 12B), match the longer queues together and fill the gap. Repeat Step 9.

9B.2. Otherwise, assign the task to the second processor. Repeat Step 9.

10. If no subtree remains, the assignment is complete. Stop.

11. If one subtree remains to be assigned, we have an assignment break. Set aside the schedule and begin Algorithm 3.5 at Step 2 with the subtree.

12. If an assignment stop has occurred,

12 A. If an assignment gap remains unfilled, match the longer

queues together and fill the gap.

12B. Assign tasks to the second queue not to exceed the length of the longer queue. This creates a possible <u>assignment</u> stop gap.

12C. Set <u>level</u> equal to <u>level</u>-1. Go to Step 5.

13. Set <u>level</u> equal to <u>level</u>-1. Go to Step 6.

The complete schedule consists of the successive partial schedules derived in Algorithm 3.5.

Now we must verify that Algorithms 3.4 and 3.5 do indeed produce an optimal schedule for Problem 3.1.

<u>Theorem 3.1</u>. Algorithm 3.5 and 3.4 find an optimal schedule for Problem 3.1.

Proof.

The strategy of the proof is to divide the schedule into segments each of which is independent of earlier segments and later segments. In this way no task from one segment may be assigned in an earlier segment.

Algorithm 3.5 concludes portions of the schedule in several places. These are Step 4, Step 7F, Step 10, and Step 11. Step 3D calls upon Algorithm 3.4 to correct the schedule if possible. We show that schedules achieved at each termination are optimal.

Step 4 is executed only when the current level is level M. In this step a single subtree exists in which there is only one task, t, such that for all tasks $x \in G$, $x \neq t$, $x \prec t$. Since task t is indivisible, only one processor may execute t while the second processor remains idle. We call the queue to which t is assigned the <u>longer queue</u>. The partial optimal schedule consisting only of t is

set aside. After removing t from the set of tasks the algorithm is reentered at Step 2 to find the schedule of the remaining tasks.

Step 7F, Step 10, and Step 11 are similar. In all these Steps an assignment break occurs before a current level is encountered which has three or more tasks. In Step 10 the assignment is complete; in Step 11 all remaining tasks form a single subtree consisting of a task t and tasks x, $x < t$.

<u>Lemma 3.1</u>. At time T, t_1 is executed on processor 1, t_2 is executed on processor 2, and $L(t_1) < L(t_2)$. At time T', $T' > T$, t_1' is executed on processor 1, t_2' is executed on processor 2, and $L(t_1') > L(t_2')$. Then at some time T'', $T \le T'' \le T'$, t_1'' is executed on processor 1, t_2'' is executed on processor 2, $L(t_1'') = L(t_2'')$.

Proof.

Assume that the lemma is not true. Then at some time T, $L(t_1) < L(t_2)$ and at time T+1, $L(t_1') > L(t_2')$. Algorithm 3.5 requires that $L(t_1') \le L(t_2')$.

A. Assume that $L(t_2) = L(t_2')$.

From the assumptions, $L(t_1') - 1 \ge L(t_2')$ and $L(t_1) + 1 \le L(t_2)$.

Then $L(t_1) + 1 \le L(t_2) = L(t_2') \ge L(t_1') - 1$.

This implies $L(t_1) + 2 \le L(t_1)$.

B. Assume that $L(t_2) = L(t_2') + 1$.

From the assumptions, $L(t_1') \ge L(t_2') + 1$ and $L(t_1) + 1 \le L(t_2)$.

Then $L(t_1) + 1 \le L(t_2) = L(t_2') + 1 \le L(t_1')$.

This implies $L(t_1) < L(t_1')$ which contradicts the fact that $L(t_1') \le L(t_1)$.

Lemma 3.2. If an assignment break or an assignment stop occurs at time T, each task in the longer queue is either a predecessor or a successor of each other task.

Proof.

Let $t_1, t_2, \ldots, t_r$ be the tasks assigned to the longer queue. Let $t'_1, t'_2, \ldots, t'_s$ be the tasks assigned to the second queue. Suppose at some first time T', $t_{i-1} \sim t_i$. $t_1 \succ t_2 \succ \ldots \succ t_{i-1} \succ t_i$. t_i must have been assigned during Step 9 of Algorithm 3.5 and $L(t_i) = L(t_{i-1})$. If t'_j is the task assigned to the second processor at time T', $L(t_i) < L(t'_j)$. Since $L(t_i) = L(t_{i-1})$, the condition $L(t_i) = L(t'_j)$ would require that $L(t_i) = L(t_{i-1}) = L(t'_j)$. This situation would have been detected as $N(L(t'_j)) = 3$ in Step 7 of Algorithm 3.5.

Since an assignment break occurs at time T, at least one task remains to be assigned. $L(t_r) > 1$ and $L(t'_s) = 1$. Therefore, $L(t_r) > L(t'_s)$.

By Lemma 3.1, at some T'', $T' < T'' < T$, $L(t) = L(t')$.

A. If $t_1 \succ t$, then since in Step 9 tasks from the dedicated chain are assigned first, there exists a task t^*, $t^* \prec t_i$ for which $L(t^*) = L(t) = L(t')$. Hence we again have $N(L(t)) \geq 3$ which Step 7 of Algorithm 3.5 would have detected.

B. If $t_1 \sim t$, then t is the last task at $L(t)$. If there were more, we again would have $N(L(t)) \geq 3$. How many tasks are in level $L(t) - 1$? There are at least a predecessor of t_1, a predecessor of t, and a predecessor of t'. Also there may be no tasks at level $L(t) - 1$. Since there is an assignment break occurring later, both situations lead to a contradiction.

Lemma 3.3. If an assignment break occurs in a partial assignment which contains assignment stops, each task in the longer queue is either a predecessor or a successor of each other task.

Proof.

In Step 12 of Algorithm 3.5 the assignment stop is detected. All tasks in the longer queue are related by Lemma 3.2. Step 7B, Step 9B.1, and Step 12A unite subsequent portions of the schedule at the assignment stop by matching longer queues. Since the first task of each queue in the subsequent portion is related to the tasks in the longer queue at the assignment stop, the tasks in the longer queue of the united portions are related. ∎

The set of tasks up to the assignment break begins on both processors simultaneously. No idle time occurs on one processor while the second is executing some tasks except at the end. Since all tasks in the longer queue are related, no shorter time for completion of this set of tasks is possible. Since all succeeding tasks, if any, are predecessors of the last task assigned to the longer queue, no tasks left unassigned may be included in this portion of the schedule.

Step 3D calls upon Algorithm 3.4 to complete each schedule in which some current level has three or more tasks. By Lemma 3.1 a schedule in which no current level has three or more tasks is such that each task in the longer queue is a predecessor or successor of each other task. Therefore, no shorter schedule is possible.

Unless $N(M) \geq 3$, Step 7 detects that a level with three or more tasks is next to be scheduled and the longer queue is processing tasks

while the other processor is idle for no more than two units. First, a stop gap may remain unfilled. Since a stop had occurred and not an assignment break, the situation is the same as in Lemma 3.1. If N(level-1) < 3 after the stop gap is filled, the algorithm continues as if N(level-1) had never been greater than or equal to 3.

If no stop gap remained or if N(level-1) ≥ 3 after the stop gap was filled, Steps 7B through 7F are used. Since an assignment break does not exist, sufficient tasks exist to satisfy Step 7B. If levels (level-1) or (level-2) have less than 3 tasks after Step 7B, or both, one queue may receive one set and the second queue the other. At most two units of deficiency exist. Then execution goes to Step 3B.

Step 3 is entered in two ways. If N(M) ≥ 3, Step 3 is entered directly from Step 2. Steps 4 through 13 are never executed. Otherwise, Step 3B follows Step 7. In either case Step 3B is entered in the following situation.

a. Form A: Either level M was of Form A or Step 7E left Form A. In both cases, the last task assigned to both queues is from the previous level. The current level has three or more tasks.

b. Form B: Form B may be the form of level M or Form B may be left by Steps 7B, 7D, or 7E. In all cases, the current level has three or more tasks.

c. Form C: Form C may be the form of level M or Form C may be left by Steps 7B, 7D, or 7E. In all cases the current level has three or more tasks.

For this discussion if N(M) ≥ 3, level k is level M. Otherwise, level k refers to the current level in a, b, and c, above, which is

the first level with three or more tasks. Given the partial assignment through level k, each additional level is added using Algorithm 3.2 by Steps 3C.2 through 3C.4. The results of these assignments are summarized in Table 3.1 which is easily verified by case analysis. Consequently the resulting schedule at Step 3D may be of Form A, Form B, or Form C.

Lemma 3.4. No idle time occurs in a partial schedule by Algorithm 3.5 except possibly on one processor at the end of the schedule.

Proof.

Tasks in Algorithm 3.2 and in Algorithm 3.5 at Steps 3, 4, 7, and 9 are scheduled without delay between tasks. In Step 12 of Algorithm 3.5 a stop gap is allowed to exist. However, by definition sufficient tasks are available to fill the gap. The tasks are assigned when the longer queues are matched in Step 7A, 9B.1, and 12A.

Since no idle time occurs in the partial assignment by Lemma 3.4, schedules of Form B and Form C are optimal by Corollary 3.5. Therefore, only assignments of Form A must be shown to be optimal or transformed to Form C by Algorithm 3.4.

a. If the length of the longest queue is equal to the length of the longest path in the subtree, the Form A is optimal.

b. If the assignment contains no 1-unit tasks, there are an odd number of 2-unit tasks. No better schedule may be obtained.

The 1-unit tasks occur either in levels M, M-1, ... , k+1 or in levels k, k-1, ... , 1 where k is the largest numbered level such that $N(k) \geq 3$ found in Step 2.

c. If a 1-unit task occurs in levels k, k-1, ... , 1, Step 3 finds the smallest value i such that level i contains a 1-unit task.

1. $k \geq i$, the partial assignment up to level i+1 is of Form B and level i is of Form B by Algorithm 3.2: By Table 3.1, (Form B)·(Form B) gives Form C. By moving one 1-unit task from one processor to the next in Step 3A, the form becomes Form A. By Table 3.1 the assignment of levels of Form C and Form A which before created Form A now creates Form C. Since all remaining levels have only 2-unit tasks, a complete assignment which before was Form A now becomes Form C by Table 3.1.

2. $k \geq i$, the partial assignment up to level i+1 is of Form C and level i is of Form C by Algorithm 3.2: Again one of the 1-unit tasks can be moved from one processor to the second or done in Step 3B. The form which was (Form C)·(Form C) = Form C now becomes Form A. By Table 3.1 and the fact that no 1-unit tasks occur later, the assignment is completed as Form C.

3. $k > i$, the partial assignment up to level i+1 is of Form A, and level i is of Form C by Algorithm 3.2 with $N(i) > 3$. Since $N(i) > 3$, there are more than one task other than the two 1-unit tasks. At least one of these tasks may be assigned during the partial assignment deficit. This leaves at least one 1-unit task which may be moved to the other processor to create Form A instead of Form C as in Step 3B.

4. $k > i$, level i is of Form C by Algorithm 3.2 with $N(i) = 3$ and the partial assignment up to level $i+1$ is of Form A. Step 1C finds the largest number j such that $k > j$, $N(j) = 3$, and level j is of Form C by Algorithm 3.2. If j does not exist, the assignment length equals the length of the longest path in the tree. The last tasks assigned in level $j+1$ are unrelated, and two do not conflict with the 2-unit task of level j. By assigning one of these two tasks last in level $j+1$, as in Step 3C, the partial assignment up to level j becomes Form C instead of Form A.
5. $k \geq i$, level i is of Form C by Algorithm 3.2, $N(i) = 3$, $N(i-1) \geq 3$, and the partial assignment up to level i is of Form A. Level $i-1$ contains only 2-unit tasks. Two of which, respectively, are predecessors of the 1-unit tasks of level i. As in Step 3D, by placing a 1-unit task t of level i before the 2-unit task of level i, the predecessors of the other 1-unit task t' of level i may be assigned in place of t. Then the predecessor of t is assigned after that. All other tasks of level $i-1$ are assigned as usual. If the original form of the partial assignment was Form A, or Form C, it now becomes Form C or Form A, respectively. By Table 3.1, the complete assignment becomes Form C.
6. No 1-unit tasks exist in levels k, $k-1$, ... , 1. We desire to change the Form A or Form C of the partial assignment of levels M, $M-1$, ... , k to Form C or Form A,

respectively. This operation is accomplished by moving a 1-unit task from one queue to the other.

6A. In Step 4A, a last assigned 1-unit task D exists which is not in the longer queue. At the same time a task F is executing on the longer queue. Also since $N(k) \geq 3$, there exists a task E, $E \sim D$, $E \sim F$. E is a 2-unit task. D is replaced by E, and D is placed before F if F begins execution at the same time as D. Otherwise, D is placed immediately after F. Since D, in its new place, does not begin execution earlier than before, it has no conflict with earlier tasks. Since no task related to D is executed while D is executed, no conflict exists there. If E had already been assigned to the second queue, all tasks assigned before it begin no earlier than they had before. This creates no problems since no assignment fork occurred. Scheduling the tasks of level i creates a different form than before.

6B. In Step 4B, one extra unit is again assigned to the longer queue with the same results as in Step 6A.

d. Now it must be shown that all possible occurrences of non-optimal schedules have been corrected.

1. A schedule which ends in Form A and whose last level i with 1-unit tasks is of Form B, has a partial schedule through level i+1 of Form B. Otherwise, a Form B would

result after assignment of level i which could not be changed to Form A by the tasks in succeeding levels of Form A and Form C. This case is taken care of in paragraph 1 of c above.

2. A schedule which ends in Form A and whose last level i with 1-unit tasks is of Form C may have a partial schedule through level i+1 of Form A or Form C. Form B may not exist if the whole schedule ends in Form A. These cases are found in 2, 3, 4 and 5 of c above.

3. The last level with 1-unit tasks may not be of Form A by definition.

4. The procedure of paragraph 6 in c above succeeds only if no assignment fork occurred after the 1-unit task. Suppose a change may be made before the assignment fork at time F. Then we must increase the longer queue by one unit and decrease the shorter queue by one unit. It is impossible to decrease the longer queue since all its tasks are related. Originally, the last task t in the longer queue terminated at time T, and no tasks were assignable to the shorter queue after time T+1. Hence, a predecessor t' of t had to be assigned at time T or T+1 on the shorter queue if it were available. By increasing the longer queue by 1-unit the last task s, $s \neq t$, in general, ends at time T+1 on the longer queue. t' then must be assigned at time T-1 if $s \neq t$ or at time T if $s = t$. In both cases t, the successor of t', has not been completed, and the assignment is not possible.

3.3.1. **Analysis of Algorithms 3.4 and 3.5**

The analysis of Algorithms 3.4 and 3.5 is more involved than for Algorithm 3.1 and 3.2. But the description below shows that the solution to the tree precedence problem requires $O(n)$ operations. We begin with Algorithm 3.5.

In Step 1 the use of Algorithm 3.1 one time requires $O(n)$ operations. Its use is never repeated throughout the algorithm. Step 2 requires at most one operation per level to keep track of the current level number. In general, much less than n operations is required.

Step 3 is a complicated step which in some cases may perform the whole sequencing operation. As shown earlier the use of Algorithm 3.2 in Step 3A and Step 3C.2 is an operation which requires on the order of the number of tasks in the level for completion. The repeated use through each level would give $O(n)$ operations. Steps 3B and 3C combine, again, to require $O(M)$ operations ($M \ll n$, in general). The inspection of the current schedule needed in Step 3C.1 at each level also means $O(M)$ operations. Step 3C.3 means that each task in a level may have to be scanned to rearrange the current level schedule. In general, not all tasks must be visited here. Yet the step when repeated for each level gives $O(n)$ operations. The total number of operations in all for Step 3 is of $O(n)$.

The examination of Step 4 is a process which may require looking at two terminal tasks in the yet unscheduled tree. However, in practice the number of tasks remaining at given levels is a value the implementer would probably maintain in a separate table. In either case the number of operations for inspection is $O(n)$ through the whole procedure. If the assignment is performed by Step 4 it is of the

simplest nature and requires one operation for each use.

Step 5 is similar to Step 4 in that at most two tasks must be scanned each time the step is encountered which may be M times. But again this requires $O(n)$ operations. In Step 6 we have the same analysis.

Step 7 has many options. In Step 7A some mechanism is required to retain the fact that a stop gap is unfilled. Several, perhaps all, currently assignable tasks may be visited. In the worst case this may be nearly all n tasks. However, Step 7 may be used only once during an assignment. A practical implementation on the other hand may apply Step 1 of Algorithm 3.2 to all levels before going beyond Step 3 to facilitate these searches. Step 7B is similar to Step 7A and may use information obtained by its search so as not to repeat unsuccessful attempts. Step 7C is a test and level count update if used. Step 7D, 7E, and 7F perform simple two task assignments. Then either the assignment is complete or Step 3 is entered. For Step 7 the number of operations is of $O(n)$ for its single use.

Steps 8 through 13 are lengthy to describe but computationally simple. Step 8 requires a simple test of a variable constantly updated. Step 9 may be repeated several times at each use. However, the value of next level would be constantly maintained in an implementation to facilitate its use. Although the step may be repeated more than once, it is not repeated more than the number of levels. Usually the number is much less than M. The searches in both 9A and 9B may require looking at nearly all tasks. From the complete algorithm the number of operations is about Mn.

Step 10 and Step 11 inspect the remaining subtree and require knowing the number of tasks awaiting assignment in the next highest level.

Step 12A is analyzed in the same manner as 7A. Step 12A and 12B may require looking at the small number of tasks awaiting execution. Steps 12C and 13 require merely bookkeeping for the level count. The number of operations is much less than n.

Algorithm 3.4 is applied in certain cases to the resulting schedule after Step 3 of Algorithm 3.5 is used. Step 2 requires locating a special level which is easily maintained in an implementation of Algorithm 3.5.

In Step 3 the scan of the levels cannot require looking at more than each task once. However, again an implementation may simplify this process by some bookkeeping. Steps 3A, 3B, 3C, and 3D are local fixups and require looking at a few tasks in at most two levels or finding another specific level. Again no task must be looked at more than once. To complete the optimal schedule Step 3 of Algorithm 3.5 as embodied in Table 3.1 must be applied. In all the number of operations is $O(n)$.

The analysis of Step 4 is similar to that for Step 3. Again a search for specific tasks looks at each task at most once. The transformation is local although several tasks assigned later may be inspected. Again the optimal schedule is completed using Table 3.1. In all the number of operations is $O(n)$.

The practical use of the algorithm occurs when $M \ll n$. In these cases the number of operations when summed throughout the algorithm is of $O(n)$.

3.4. A Solution to the Tree-restricted Acyclic Precedence Problem

A more difficult problem is an extension of the tree precedence problem to 1-unit and 2-unit tasks with acyclic precedence. Its solution would be a major step forward in scheduling research. Unfortunately, we

have been unable to find an efficient solution. Instead, the special case defined by Problem 3.2, the tree-restricted acyclic precedence problem, is considered here. First, we repeat the problem statement from Section 3.1.

Problem 3.2. The Tree-restricted Acyclic Precedence Problem

Find an optimal two-processor schedule for a set G of n 1-unit and 2-unit tasks with p tree-restricted acyclic precedence.

Algorithm 3.6 presents a procedure to solve the tree-restricted acyclic precedence problem. The method is repeated applications of the solution to the tree precedence problem. This process is successful because each tree in a set G must be completed before any tasks in the successive tree may begin.

Algorithm 3.6.

1. Locate the p sets of tasks with tree-precedence, A_1, A_2, ... , A_p, such that A_{i-1} is maximally connected to A_i, $i = 2, 3, \ldots, p$.
2. Schedule each set of tasks with tree-precedence, A_1, A_2, ... , A_p, using Algorithms 3.4 and 3.5.
3. Execute the sets of tasks in order A_1, A_2, ... , A_p.

Theorem 3.2. Algorithm 3.6 produces an optimal schedule for n 1-unit and 2-unit tasks with p tree-restricted acyclic precedence on two processors ($m = 2$).

Proof.

By Theorem 3.1 each set of tasks, A_i, $i = 1, 2, \ldots, p$ with tree precedence is scheduled optimally by Algorithms 3.4 and 3.5.

For all tasks x in A_{i-1} and all tasks y in A_i, $x < y$, for $i = 2, 3, \ldots, p$, by Definitions 3.1 and 3.2. All tasks in A_{i-1}

must be completed before any task in A_i may begin execution. Since for all i, A_i is scheduled optimally, the whole schedule is optimal.

3.5. A Look Ahead

The central result presented in this chapter is an algorithm for the optimal scheduling on two processors of 1-unit and 2-unit tasks with tree precedence. Our work has not produced an extension of these results to the problem for two processors and 1-unit and 2-unit tasks with acyclic precedence. However, in the latter problem the technique of subschedules may be applicable. Also the structuring of schedules so that all tasks in one section must be completed before any other task begins has been successful here and in the Coffman and Graham solution. This technique may also facilitate the solution of the general problem.

In addition to the problem suggested in the previous paragraph the problem of finding an optimal schedule on three or more processors for 1-unit tasks with acyclic precedence remains unsolved. We feel that both these problems should be studied at this time. This area suffered eleven years without major results between Hu's solution and the Coffman and Graham solution. The repetition of eleven years with no results, we feel, is unlikely.

Chapter 4

Segmented Processor Scheduling

4.1. The Segmented Scheduling Problem

This chapter presents work concerning special cases of the two machine ($m = 2$) problem with n tasks. The problem is formally presented in Problem 4.1. Informally, however, the reasonableness of the restrictions are readily apparent.

In computer scheduling it is sometime advantageous to queue a group of tasks (programs) which use the same facility (compiler) which is serially reuseable (core resident). In this case intermixing a queue of dissimilar tasks would cause set up delays of disproportionate length. Similarly, the processing (execution) of these tasks on a second machine may also require special facilities (run time administration) which are also serially reuseable. Finally, the completion of the task processing (output) may again be performed by the first processor with advantages of grouping. Problem 4.1 establishes these requirements.

<u>Problem 4.1</u>. Segmented Scheduling Problem

Find a schedule for n tasks composed of three operations. The first and third operations of each task must be performed on Machine One and the second operation must be performed on Machine Two ($m = 2$). Using Jackson's notation of Section 2.2, the tasks are divided into two sets: {ABA} and {BAB}. The form of the solution is restricted as described below to a schedule with no idle time.

Machine One: The initial operations of the set {ABA}, followed by the second operations of the set {BAB}, and followed by the third operations of the set {ABA}.

Machine Two: The initial operations of the set {BAB}, followed by the second operations of the set {ABA}, and followed by the third operations of the set {BAB}.

The segmented scheduling problem restricts the solution to one of the four forms illustrated in the Gantt charts [Clark 1947] of Figure 4.1. In these charts the time on each processor is divided into segments and labelled with the set tasks to be assigned in that segment. An underline within the brackets indicates the operations for the set of tasks which is to be performed in the particular segment. For example, {ABA̲} indicates that the third operations of the tasks in the set {ABA} are processed.

By the symmetry of Machine One with respect to Machine Two, Gantt charts II and III are similar, and Gantt charts I and IV are similar. The discussion will be limited, therefore, to forms I and II.

The following sections of this chapter contain results of several subproblems of the segmented scheduling problem. First, in Section 4.2 we limit our consideration of the problem to those cases which have the form of Gantt chart II. We call this restricted problem the "special segmented problem". Then by limiting the operation lengths of the tasks in this new problem we define and solve new subproblems. These problems are:

a. The special segmented problem with tasks having 1-unit and 2-unit operations. (Section 4.2.2)

b. The special segmented problem with tasks having successive operations that differ by one unit in length and that differ by k units in length. (Section 4.2.3)

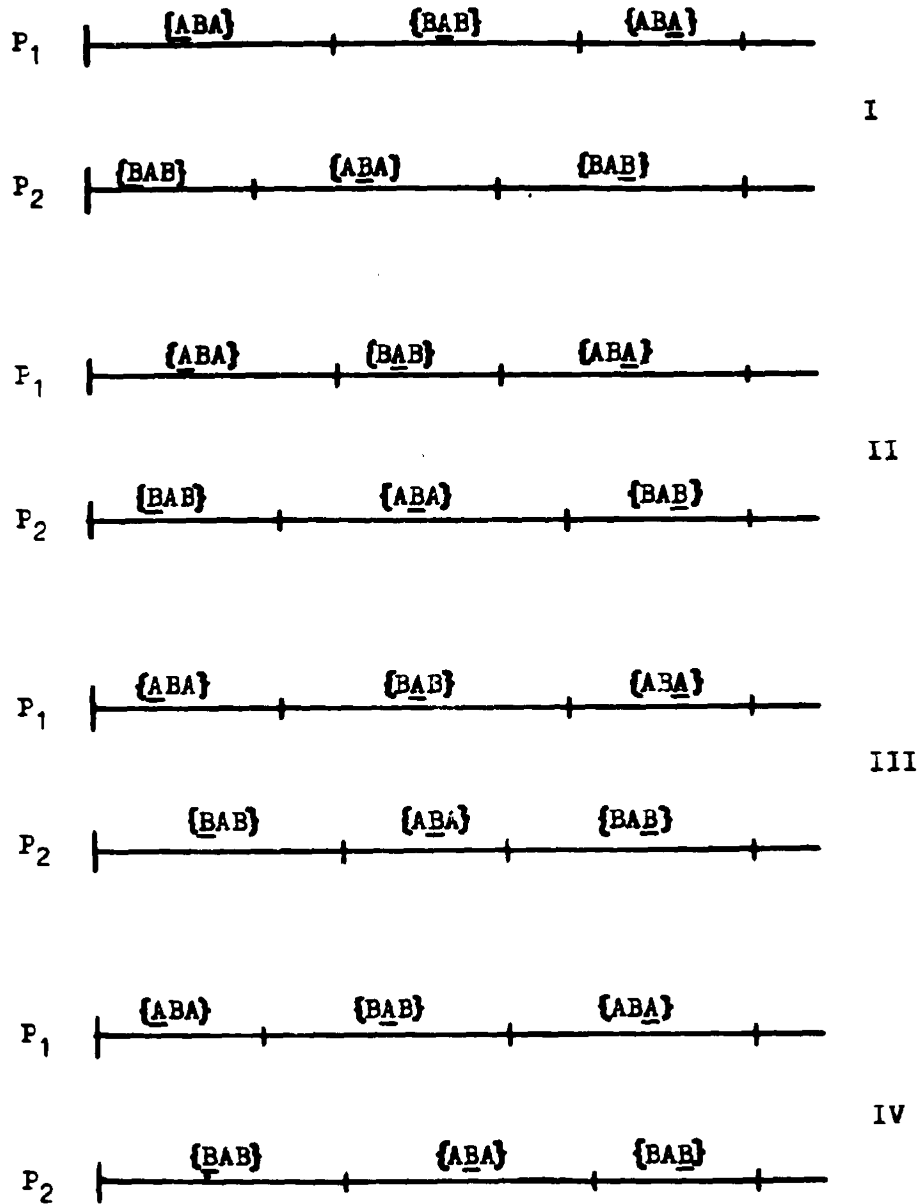


Figure 4.1. Forms of the Special Segmented Problem

c. The special segmented problem with tasks having first and third operations of equal length. (Section 4.2.4)

Finally, the special segmented problem is extended to four segments. We then solve the new problem when the tasks have first and last operations of equal length and have second and third operations of equal length. (Section 4.2.5)

Before we proceed, however, it is worthwhile to consider the relationship of our problem to the general m X n sequencing problem. As mentioned above, the segmented scheduling problem reflects certain constraints found in some computer scheduling problems. But the solution to the problem as restricted by these conditions is not necessarily an optimal solution for the general m X n sequencing problem. Figures 4.2 and 4.3 show examples in which no solution satisfying the conditions of the segmented scheduling problem may be found.

4.2. Segmented Scheduling Problem

When we consider problems which have the form of Gantt chart I, the problems have a very simple solution. The reason for the ease of solution is that the operations are decoupled.

Definition 4.1. Two successive operations in a set of tasks in the segmented scheduling problem are decoupled if all of the first operations of all the tasks in the set can be completed before any of the successor operations of any of the tasks in the set may be initiated.

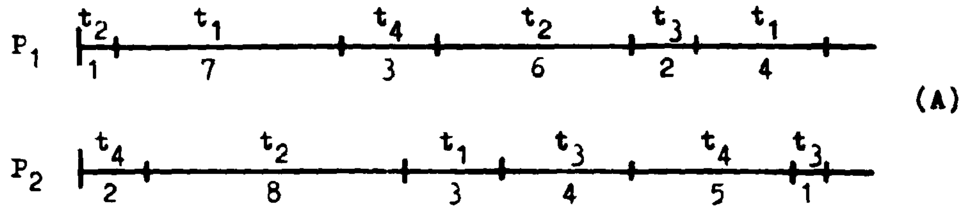
In an assignment of the type of Gantt chart I, two pairs of operations are decoupled: the first and second operations of the set {BAB} and the second and third operations of the set {ABA}. The order in which the first operations of the set {BAB} are performed, therefore,

Given the four tasks

$$\{ABA\} = \{t_1 = (7,3,4),\ t_2 = (1,8,6)\}$$

$$\{BAB\} = \{t_3 = (4,2,1),\ t_4 = (2,3,5)\}$$

an assignment may be found which contains no idle time and is completed in 23 time units.



The ordering requirement on each processor may not be maintained without adding idle time. The best solution that satisfies the ordering condition on both processors is an assignment of length 24.

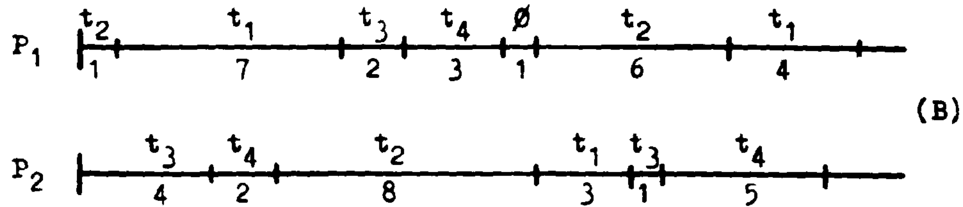
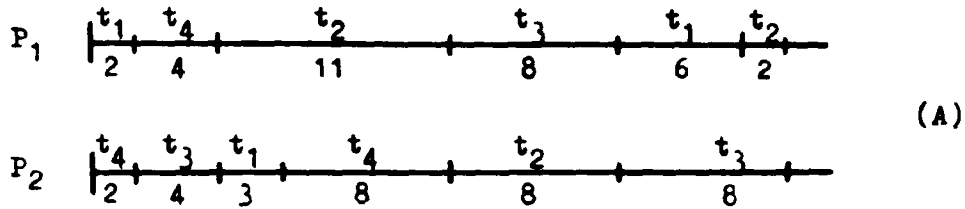


Figure 4.2. No solution to Problem 4.1 in Form II

Given the four tasks

$$\{ABA\} = \{t_1 = (2,3,6),\ t_2 = (11,8,2)\}$$
$$\{BAB\} = \{t_3 = (4,8,8),\ t_4 = (2,4,8)\}$$

we obtain the minimal solution of length 33.



The ordering requirement on each processor may not be maintained without adding idle time. The best solution that satisfies the ordering condition on both processors is an assignment of length 37.

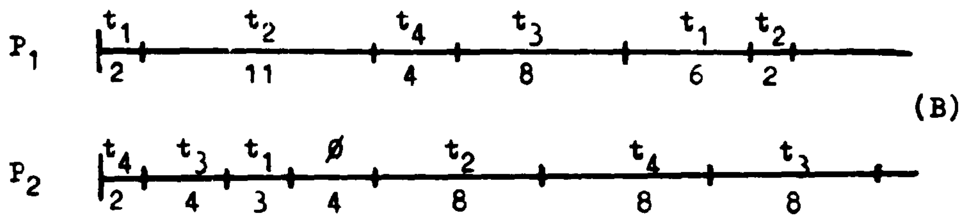


Figure 4.3. No solution to Problem 4.1 in Form I

is arbitrary. Likewise, the order in which the third operations of the set {ABA} are performed is also arbitrary. The remaining operations may be assigned using Johnson's method if a feasible assignment is at all possible with this form.

The form of the problem characterized by Gantt chart II poses a more challenging problem. Clearly, the operations of {BAB} may be performed without any regard to their relative order since both pairs of successive operations are decoupled. We are then concerned only with the assignment order of the operations of tasks in set {ABA}. To facilitate later discussion we define this problem as the Special Segmented Problem.

Definition 4.2. The Special Segmented Problem is that special case of Problem 4.1 which is characterized by Gantt chart II.

4.2.1. A Foundation for New Results

The results of Bauer and Stone [1970] show that several subproblems have efficient solutions. However, one core problem remains unsolved. In this section these solutions are given as background for new results. First a few basic definitions are required.

Definition 4.3. A stage i of a given machine is a segment of time in which the i-th operations, and only the i-th operations, of all tasks are scheduled.

Definition 4.4. A delay, $\Delta_{i,j}$, is the difference between the time a task's j-th operation is initiated and its i-th operation is initiated.

Definition 4.5. The gap is the segment of time after the first stage terminates and the third stage initiates.

Each task, t_k, k = 1, 2, ... , n, consists of three operations a_k, b_k, and c_k to be scheduled in stages 1, 2, and 3, respectively. Then

we may define the contribution a task makes to the delay.

Definition 4.6. Let x_i, y_i be a pair of successive operations of a task t_i. Then the contribution $C(x_i,y_i)$ of task t_i is the difference $y_i - x_i$.

It is important to know if a task is assignable without causing delays in the schedule during which no operation may be executed. The following definition determines the condition of assignability.

Definition 4.7. A task t_i is immediately assignable if

$a_i \le \Delta_{1,2}$ and $b_i \le \Delta_{2,3}$.

If for each task t_k, $k = 1, 2, \ldots, n$, $a_k \le b_k \le c_k$, then the contributions $C(a_k,b_k)$ and $C(b_k,c_k)$ are nonnegative. As soon as a task becomes immediately assignable, it may be assigned. In no case may the task reduce the values of the delays, $\Delta_{1,2}$ and $\Delta_{2,3}$. If a task or group of tasks never becomes immediately assignable, the problem has no solution.

The second problem consists of tasks t_k, $k = 1, 2, \ldots, n$, where $a_k \ge b_k \ge c_k$. The problem is identical to that described in I when the following transformation is performed.

Definition 4.8. The mirror image problem is the problem obtained by two transformations of the original problem.

1. The precedence among the three operations is reversed. For example, with $a_k < b_k < c_k$ in the original problem, $c_k > b_k > a_k$ in the mirror image problem.
2. The initial delays $\Delta_{1,2}$ and $\Delta_{2,3}$ in the original problem become $\Delta_{3,2}$ and $\Delta_{2,1}$, respectively.

The third problem consists of tasks t_k, $k = 1, 2, \ldots, n$, such that $a_k \ge b_k$ and $c_k \ge b_k$. This problem has the same characteristics as its mirror image problem. In brief, Bauer and Stone give a solution

consisting of a functional equation. The solution depends upon the fact that the tasks may be divided into two groups. In each group one operation of all the tasks are decoupled from the other two operations. The tasks of each group of tasks are found in the order they must be assigned. They are selected from a list of tasks arranged by the increasing size of the second task.

The problem consisting of tasks t_k, $k = 1, 2, \dots, n$, such that $a_k \le b_k$ and $c_k \le b_k$ is unsolved by an efficient solution. We call this the core problem. In succeeding sections special cases of this problem are discussed.

4.2.2. A Special Segmented Problem

Problem 4.2.

Find a schedule for n tasks composed of three operations. The first and third operations must be performed on Machine One, and the second operation must be performed on Machine Two ($m = 2$). The length of each operation is 1-unit or 2-units. The form of the schedule is restricted to the Special Segmented Problem with no idle time.

The restriction upon the length of each operation in Problem 4.2 leaves only eight possible task forms.

111 112 121 122

211 212 221 222

The contributions $C(a_k,b_k)$ and $C(b_k,c_k)$ may equal only -1, 0, or +1 for each task k, $k = 1, 2, \dots, n$. Algorithm 4.1 schedules the tasks in an order which assures a correct assignment if one exists. If no optimal assignment exists for Problem 4.2, the algorithm fails.

The main strategy of Algorithm 4.1 depends upon the delay between successive operations. One delay is usually too small to accommodate the task with the maximum size operation corresponding to that delay. Hence, it is desirable to build up the value of the delay to accommodate this maximum task by assigning immediately assignable tasks with the largest contributions to offset the deficient delay. This process is repeated upon the delay that is deficient until all tasks are assigned, if that is possible.

Algorithm 4.1.

1. Divide the tasks into four sets

   Set I = {tasks of forms 111, 112, 122, 222}

   Set II = {tasks of form 212}

   Set III = {tasks of form 121}

   Set IV = {tasks of forms 211, 221}

2. Calculate initial values of $\Delta_{1,2}$, $\Delta_{2,3}$, $\Delta_{3,2}$, and $\Delta_{2,1}$.

3. Assign all unassigned tasks from Set I which are assignable. Update values of $\Delta_{1,2}$ and $\Delta_{2,3}$. If Set II = $\emptyset$ and Set III = $\emptyset$, go to Step 12.

4. If $\Delta_{1,2} > 2$ and $\Delta_{2,3} \geq 1$ and Set II $\neq \emptyset$, assign task from Set II. Update $\Delta_{1,2}$ and $\Delta_{2,3}$. Go to Step 3.

5. If $\Delta_{1,2} > 2$ and $\Delta_{2,3} \geq 1$ and Set II = $\emptyset$, assign task from Set III. Update $\Delta_{1,2}$ and $\Delta_{2,3}$. Go to Step 11.

6. If $\Delta_{1,2} = 2$ and $\Delta_{2,3} = 1$, assign task from Set II. Update $\Delta_{1,2}$ and $\Delta_{2,3}$. Go to Step 3.

7. If $\Delta_{1,2} = 1$ and $\Delta_{2,3} > 2$ and Set III $\neq \emptyset$, assign task from Set III. Update $\Delta_{1,2}$ and $\Delta_{2,3}$. Go to Step 3.

8. If $\Delta_{1,2} = 2$ and $\Delta_{2,3} > 2$ and Set III $\neq \emptyset$, assign tasks from

Set III. Update $\Delta_{1,2}$ and $\Delta_{2,3}$. Go to Step 3.

9. If $\Delta_{1,2} = 2$ and $\Delta_{2,3} > 2$ and Set III = $\emptyset$, assign task from Set II. Update $\Delta_{1,2}$ and $\Delta_{2,3}$. Go to Step 11.

10. Assignment fails.

11. If Set II $\neq \emptyset$ or Set III $\neq \emptyset$, assignment fails.

12. Assign all unassigned tasks from Set I which are assignable. Update values of $\Delta_{1,2}$ and $\Delta_{2,3}$. If Set I $\neq \emptyset$, assignment fails.

13. Reverse time. If Set IV = $\emptyset$, assignment is complete.

14. Assign all unassigned tasks from Set IV which are assignable. Update $\Delta_{3,2}$ and $\Delta_{2,1}$.

15. If Set IV $\neq \emptyset$, assignment is complete.

16. If tasks exist which were assigned before, assign last task assigned. If not, assignment fails.

17. Go to 14.

The algorithm is quite straightforward. However, a proof that its output is indeed the optimal solution desired is required. Theorem 4.1 provides such a proof as well as a discussion of the algorithm step by step.

Theorem 4.1. Algorithm 4.1 provides an optimal solution to Problem 4.2, if one exists.

Proof.

Assume that a feasible assignment is possible but that Algorithm 4.1 fails. Failure may only occur at Steps 10, 11, 12, or 16.

A. Failure occurs at Step 10 or Step 11. Assume that Set IV = $\emptyset$ since Set IV $\neq \emptyset$ can only make the situation worse.

A.1. Suppose Set II $\neq \phi$ and Set III $\neq \phi$ and Set III $\neq \phi$. If $\Delta_{1,2} > 0$ and $\Delta_{2,3} > 0$ initially, Steps 4 through 9 do not allow $\Delta_{1,2}$ or $\Delta_{2,3}$ to become 0 while Set II is empty (Step 5). If either $\Delta_{1,2} = 0$ or $\Delta_{2,3} = 0$ initially, no assignment was possible. Then when failure occurs at Step 10 or Step 11, $\Delta_{1,2} = \Delta_{2,3} = 1$ since either equaling 2 would allow a task to be assigned. Step 3 assigned all tasks of form 112 which were available. No condition in Step 4 through 9 was ever satisfied since this would have increased either $\Delta_{1,2}$ or $\Delta_{2,3}$. The latter means one was zero which contradicts the existence of a feasible assignment.

A.2. Suppose Set II $\neq \phi$ and Set III $= \phi$. When failure occurs at Step 10 or Step 11, either $\Delta_{1,2} < 2$ or $\Delta_{2,3} < 1$. Since Steps 4 through 9 do not permit either $\Delta_{1,2}$ or $\Delta_{2,3}$ to be zero, only $\Delta_{1,2} = 1$ and $\Delta_{2,3} \geq 1$ is permissible if a feasible assignment exists. If more than one task is in Set II, all contributions would reduce $\Delta_{1,2}$ to negative.

A task in Set II cannot be assigned last; if a feasible assignment exists, a task in Set III must be assigned last. Assume a task from Set III is left until last and all other tasks in Set II and Set I were assigned. The resulting values of $\Delta_{1,2}$ and $\Delta_{2,3}$ would change to $\Delta'_{1,2} = \Delta_{1,2} - 2$ and $\Delta'_{2,3} = \Delta_{2,3} + 2$. Since $\Delta_{1,2} = 1$, $\Delta'_{1,2} = -1$ which means an assignment may not leave either a member of Set II or Set II to be assigned last from the two sets. No assignment was feasible.

A.3. Suppose Set II $= \phi$ and Set III $\neq \phi$. By similar reasoning to those in part A.2, a contradiction to the existence of a

feasible assignment is found.

B. Failure at Step 12. Again Set IV may be disregarded and assumed empty. Since tasks in Set I are assigned as soon as they are feasible, either

1. $\Delta_{1,2} = 0$ or $\Delta_{2,3} = 0$, initially
2. $\Delta_{1,2} = 1$ and $\Delta_{2,3} > 1$ or $\Delta_{1,2} > 1$ and $\Delta_{2,3} = 1$, always, and a task of form 222 remains

or 3. $\Delta_{1,2} = 1$ and $\Delta_{2,3} = 1$, always, and tasks of forms 122 or 222 remain.

B.1. $\Delta_{1,2} = 0$ or $\Delta_{2,3} = 0$, initially. No feasible assignment is possible.

B.2. $\Delta_{1,2} = 1$ and $\Delta_{2,3} > 1$ or $\Delta_{1,2} > 1$ and $\Delta_{2,3} = 1$, always, and a task of form 222 remains. Then the case never occurred when $\Delta_{1,2} \geq 2$ and $\Delta_{2,3} \geq 2$. Either $\Delta_{1,2} = 1$ or $\Delta_{2,3} = 1$ at all times. This forced the assignment to be from Set III or Set II, respectively, in order that neither becomes 0. Since both Set II and Set III are empty, and only tasks of form 222 remain in Set I, no assignment is feasible.

B.3. $\Delta_{1,2} = 1$ and $\Delta_{2,3} = 1$ and tasks of form 122 and 222 remain. It was never the case that $\Delta_{1,2} = 1$ and $\Delta_{2,3} \geq 2$ or that $\Delta_{1,2} \geq 2$ and $\Delta_{2,3} \geq 2$ occurred. No task other than 111 was ever assigned since at some time either $\Delta_{1,2}$ or $\Delta_{2,3}$ would have been zero. No assignment is feasible.

C. Failure at Step 16.

C.1. If 211 tasks remain to be assigned, then either $\Delta_{3,2} = 0$

or $\Delta_{2,1} = 0$ initially. Then no feasible assignment was possible.

C.2. All 211 tasks have been assigned, but some 221 tasks remain. It has never occurred that $\Delta_{3,2} \geq 1$ and $\Delta_{2,1} \geq 2$. Then no task from Set II or Set III has even been assigned. No 211, 122, or 221 tasks have been assigned. Only 111 and 112 tasks have been assigned, and $\Delta_{3,2} = 1$ at all times. No feasible assignment was possible.

Figure 4.4 shows the use of Algorithm 4.1 to solve an example of Problem 4.2.

4.2.3. A More General Core Problem

To extend the result of the previous section we introduce a problem which relaxes the constraint on the length of the tasks. In Problem 4.3 below the lengths of successive operations are required only to differ by one unit.

Problem 4.3.

Find a schedule for n tasks composed of three operations. The first and third operations must be performed on Machine One, and the second operation must be performed on Machine Two ($m = 2$). The first and third operations have identical length, and the second operation has length one less or one greater than the first and third operations. The form of the schedule is restricted to the Special Segmented Problem with no idle time.

The solution to Problem 4.3 is similar to that for Problem 4.2. Algorithm 4.2 gives the details of the solution method while Figure 4.5 shows an example of its use.

Tasks

$t_1 = (1,1,1)$

$t_2 = (2,1,1)$

$t_3 = (1,2,1)$

$t_4 = (1,2,2)$

$t_5 = (2,1,2)$

$t_6 = (1,2,1)$

$t_7 = (1,2,1)$

Set I = $\{t_1, t_4\}$

Set II = $\{t_5\}$

Set III = $\{t_3, t_6, t_7\}$

Set IV = $\{t_2\}$

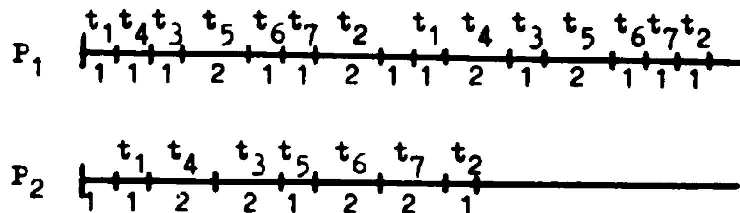


Figure 4.4. Example of Algorithm 4.1

Tasks

$t_1 = (4,5,4)$ $t_2 = (4,5,4)$

$t_3 = (5,6,5)$ $t_4 = (5,6,5)$

$t_5 = (6,7,6)$ $t_6 = (6,7,6)$

$t_7 = (7,8,7)$ $t_8 = (7,8,7)$

$t_9 = (8,9,8)$

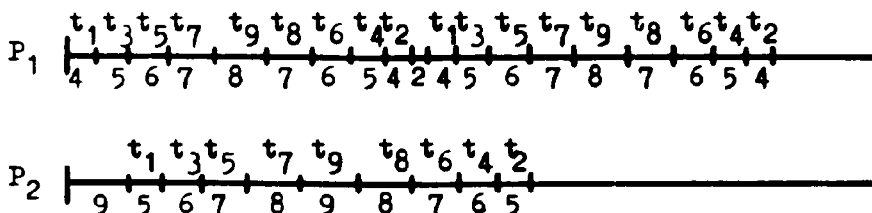


Figure 4.5. Problem 4.3 and Algorithm 4.2

If the tasks are:	Scale tasks by 2:
$t_1 = (8,10,8)$	$t'_1 = (4,5,4)$
$t_2 = (8,10,8)$	$t'_2 = (4,5,4)$
$t_3 = (10,12,10)$	$t'_3 = (5,6,5)$
$t_4 = (10,12,10)$	$t'_4 = (5,6,5)$
$t_5 = (12,14,12)$	$t'_5 = (6,7,6)$
$t_6 = (12,14,12)$	$t'_6 = (6,7,6)$
$t_7 = (14,16,14)$	$t'_7 = (7,8,7)$
$t_8 = (14,16,14)$	$t'_8 = (7,8,7)$
$t_9 = (16,18,16)$	$t'_9 = (8,9,8)$

Figure 4.6. Problem 4.4 and Algorithm 4.2

Algorithm 4.2.

1. Divide the tasks into two sets according to whether the contribution from the first and second operations is +1 or -1:

   Set I $= \{XYX \mid X+1 = Y\}$

   Set II $= \{XYX \mid X-1 = Y\}$

2. Define M_1 to be equal to the largest first operation. Define M_2 to be equal to the largest second operation.

3. Calculate $\Delta_{1,2}$ and $\Delta_{2,3}$.

4. If Set II $= \emptyset$ and Set III $= \emptyset$, the optimal schedule is complete.

5. If for some i, $i = 1$ or 2, $M_i > \Delta_{i,i+1}$, assign task with the largest valued second operation from Set I if $i = 1$ or from Set II if $i = 2$. Update values of M_1, M_2, $\Delta_{1,2}$ and $\Delta_{2,3}$. Go to Step 4. If no task may be found to assign, the assignment fails.

6. For some i, $i = 1$ or 2 and j, $j \neq i$, $j = 1$ or 2, $M_i = \Delta_{i,i+1}$ and $M_j \leq \Delta_{j,j+1}$,

   6A. If all tasks t with the i-th operation equal to M_i are in Set II, assign each task t in Set II alternately with tasks with the largest operations from Set I until all t are assigned. Update the values of M_1, M_2, $\Delta_{1,2}$, and $\Delta_{2,3}$. Go to Step 4.

   6B. If all tasks t with the i-th operation equal to M_i are in Set I, assign each task t in Set I alternately with tasks with the largest operations from Set II until all t are assigned. Update M_1, M_2, $\Delta_{1,2}$, and $\Delta_{2,3}$. Go to Step 4.

   6C. If tasks t with the i-th operation equal to M_1 are in both Set I and Set II, assign each task t in Set I followed by

a task t from Set II until all t's are assigned from either Set I or Set II. Update the values of M_1, M_2, $\Delta_{1,2}$, and $\Delta_{2,3}$. Go to Step 4.

7. If $\Delta_{1,2} \geq M_1$ and $\Delta_{2,3} \geq M_2$,

   7A. While $\Delta_{1,2} \geq M_1$ and $\Delta_{2,3} \geq M_2$, assign tasks with the largest first operation from Set I alternately with tasks with the largest first operation from Set II. Update the values of M_1, M_2, $\Delta_{1,2}$, and $\Delta_{2,3}$. Check the condition after each assignment. When the condition fails or both sets are empty, go to Step 4.

As in the case of the previous algorithm, the detailed discussion of Algorithm 4.2 is contained in the proof of a theorem. Theorem 4.2 shows that Algorithm 4.2 does produce an optimal solution for Problem 4.3.

<u>Theorem 4.2</u>. Algorithm 4.2 provides an optimal solution for Problem 4.3 if one exists.

Proof.

Assume that a feasible solution is possible to Problem 4.3, but that Algorithm 4.2 fails. Failure may only occur at Step 5. Then for all tasks XYX in Set I and Set II either

$$X > \Delta_{1,2} \quad \text{or } Y > \Delta_{2,3}$$

A. $X > \Delta_{1,2}$ and $Y > \Delta_{2,3}$: A solution was possible under the condition that $M_i < \Delta_{i,i+1}$ for $i = 1$ or 2, then for $j \neq i$, $j = 1$ or 2, $\Delta_{i,i+1} - M_i \leq M_j - \Delta_{j,j+1}$.

B. $X - \Delta_{1,2} > 1$ or $Y - \Delta_{2,3} > 1$. Then the algorithm was operating in Step 5 for the first time while attempting to build up to the

maximum task size but failed. An insufficient contribution was available to make the large task feasible. A feasible solution was not possible.

C. $X - \Delta_{1,2} = 1$ or $Y - \Delta_{2,3} = 1$. Either the algorithm was operating in Step 5 for the first time as in B, or the algorithm in Step 6 assigned a task which reduced $\Delta_{1,2}$ or $\Delta_{2,3}$ by 1 to create this situation. The first possibility leads to the same contradiction as in part B. The second possibility divides into two cases.

1. $X_1Y_1X_1$ was assigned last with $Y_1 < X_1$.
   No $X_2Y_2X_2$ was available with $Y_2 > X_2$.

or 2. $X_1Y_1X_1$ was assigned last with $Y_1 > X_1$.
   No $X_2Y_2X_2$ was available with $Y_2 < X_2$.

In either case, no task remained to make up the deficit. Hence, the wrong task was left until last. Tasks remained only in Set I or only in Set II, not both.

C1. $X - \Delta_{1,2} = 1$ and $X - Y = 1$ (Set II)

Suppose at some stage this task should have been assigned instead of another task. Whenever it was possible a task with a larger or equal length first operation was assigned.

C1.A. No task with a smaller length first operation was assigned from Set II. When a task in Set II with a smaller length first operation was assigned, XYX could not have been assigned. Assigning XYX instead of an equal or larger length task would have left that task unassigned. No switch could be made, and

the assignment was not feasible.

C1.B. If a task from Set I was replaced by XYX, $\Delta_{1,2}$ would have decreased by 2. Then another task with a + contribution would have been needed to make up the deficit. But there were no tasks in Set II with a smaller length first operation to fill this need. All were used to enable tasks with a larger or equal length first operation to be assigned. Hence, XYX could not have been used to replace a task from Set I.

C2. $Y - \Delta_{2,3} = 1$ and $Y - X = 1$ (Set I)

Arguments analagous to that in C1.

The solution to Problem 4.3 also provides a solution to a related problem. This new problem is stated in Problem 4.4. This time the constraint upon the length of the tasks operations is relaxed to include tasks with successive pairs of operations whose lengths differ by a constant. An example of the use of Algorithm 4.2 to solve this problem is in Figure 4.6.

Problem 4.4.

Find a schedule for n tasks composed of three operations a_i, b_i, c_i, $i = 1, 2, \ldots, n$. The first and third operation must be performed on Machine One, and the second operation must be performed on Machine Two ($m = 2$). The length of each operation is arbitrary, but the contribution between adjacent operations within each task must be $C(a_i,b_i) = k$, $C(b_i,c_i) = -k$ or $C(a_i,b_i) = -k$, $C(b_i,c_i) = k$, $i = 1, 2, \ldots, n$ (k is constant). The form of the schedule is restricted to the Special Segmented Problem with no idle time.

Corollary 4.1. Algorithm 4.2 provides the method of solution for Problem 4.4 if a solution exists.

Proof.

Scale each task in Problem 4.4 by dividing the length of each operation by k. That is, if a task has operations of length iji where j = i ± k, transform the task to $\frac{i}{k}, \frac{i}{k} \pm 1, \frac{i}{k}$. The contribution of each pair of operations after transformation is +1 or -1. Since the proof of Theorem 4.2 does not require operations of integral length, Algorithm 4.2 is also the method of solution for Problem 4.4 if a solution exists.

4.2.4. A Problem with a Knapsack Solution

We continue with another variation of the central problem presented earlier. The solution method differs, however, from the last several examples. In Problem 4.5, below, the constraint upon the lengths of operations is further relaxed. Here the first and third operations must have identical lengths.

Problem 4.5.

Find a schedule for n tasks composed of three operations. The first and third operations must be performed on Machine One, and the second operation must be performed on Machine Two (m = 2). The first and third operations have identical length, and the second operation has a different length. The form of the schedule is restricted to the Special Segmented Problem with no idle time.

The form of the solution to Problem 4.5 has two possibilities. References are made to terminology defined in Section 4.2.1. In one case, a decoupling point, D, occurs during the gap, G. Figure 4.7 shows a

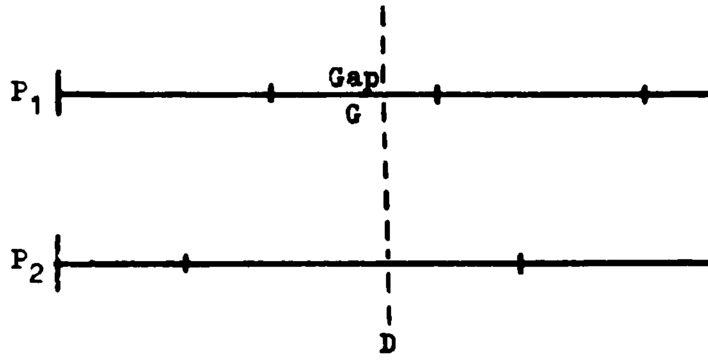


Figure 4.7. First form of Problem 4.5 solution

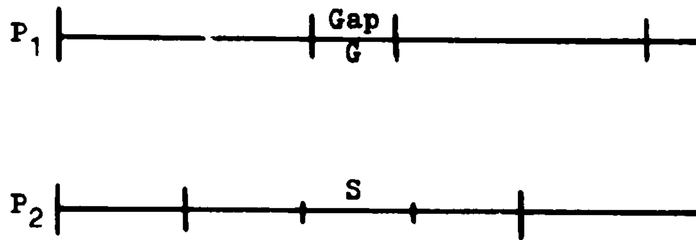


Figure 4.8. Second form of Problem 4.5 solution

characterization of this solution form. In the second case, a second operation S begins before the gap, G, and ends after the gap, G. Figure 4.8 shows the characterization for the second solution form.

The operations of each task are known to be in identical order on each of the three processors by Johnson's result. The set of tasks R occurring before the decoupling point, D, or before task S in the respective forms is decoupled between stages 2 and 3. The set of tasks T occurring after the decoupling point D or the task S, in the respective cases, is decoupled between stage 1 and 2. Hence, the R tasks may be Johnson ordered between stages 1 and 2, and the T tasks may be Johnson ordered between stages 3 and 2. However, we do not know *a priori* what tasks form sets R and T or even which task is task S.

Note immediately that the first and third operations of each task are identical in length. Consequently, the contribution $C(a_i,b_i)$ equals the contribution $C(c_i,b_i)$. This symmetry suggests the use of a two-dimensional knapsack solution described in Algorithm 4.3. Figure 4.9 shows an example of the use of Algorithm 4.3.

Algorithm 4.3.

1. Separate the tasks into two sets:

   Set I = {all tasks such that $C(a_i,b_i) \geq 0$}

   Set II = {all tasks such that $C(a_i,b_i < 0$}

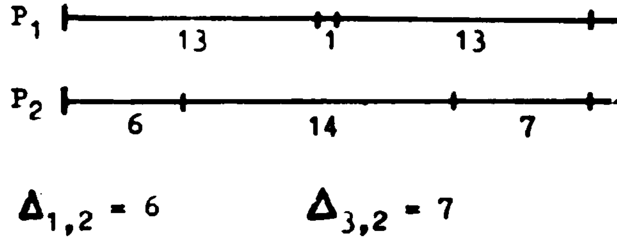
2. Transform Sets I and II into the following ordered set:

   Set III = {Set I in order of increasing size of the first operation and increasing contribution followed by Set II in order of decreasing size of the first operation and decreasing contribution loss.}

   Set $\Delta_{3,2}$ equal to the initial length from the end of stage 2 to the end of stage 3.

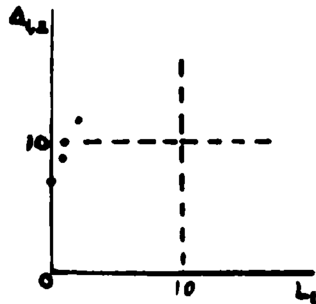
Tasks

$t_1 = (1,3,1)$
$t_2 = (1,4,1)$
$t_3 = (2,3,2)$
$t_4 = (4,3,4)$
$t_5 = (5,1,5)$

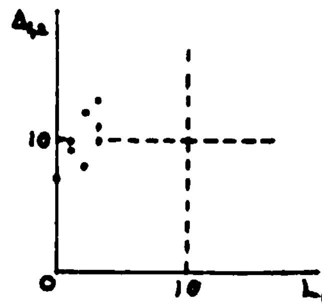


$\Delta_{1,2} = 6$ $\Delta_{3,2} = 7$

After t_1 and t_2:



After t_1, t_2, and t_3:



After all five tasks:

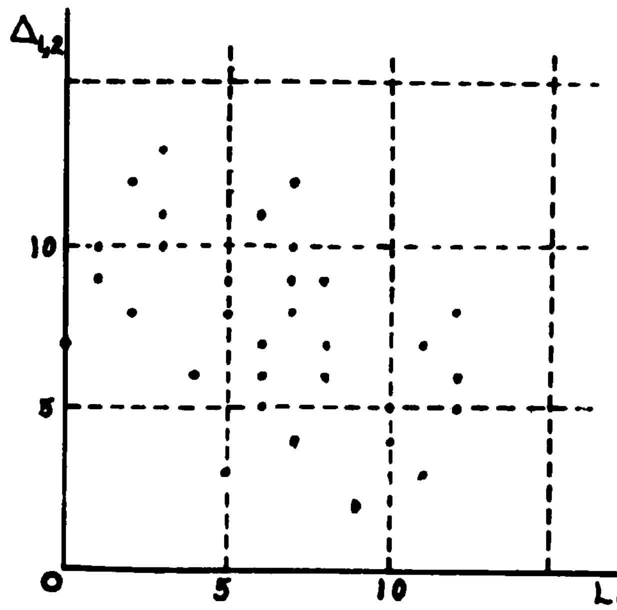


Figure 4.9. Example of Algorithm 4.3

3. Set L_1 equal to 0 -- the total length assigned to stage 1.

   Set $\Delta_{1,2}$ to the initial length of time from start of stage 1 to the start of stage 2.

   Set TL equal to 0 -- the total length of the first operation assigned.

   Set TC equal to 0 -- the total contribution $C(a_i,b_i)$ assigned.

4. Establish the first quadrant of a two-dimensional Cartesian coordinate system, $\mathcal{C}$, with the values of L_1 and $\Delta_{1,2}$ represented by the abscissa and ordinate, respectively. Mark initial values of $(L_1, \Delta_{1,2})$

5. For each task, a_i, b_i, c_i, in the ordered Set III, for each point (x,y) in $\mathcal{C}$,

   5A. if $y \geq a_i$, indicate a new point $(x+a_i, y+C(a_i,b_i))$. Go to 5B.

   5B. if $\Delta_{3,2} + (TC-y) < a_i$, delete (x,y). Go to 5C.

   5C. set $TL = TL + a_i$

   set $TC = TC + C(a_i,b_i)$

6. Select the solution corresponding to the point (x,y) where $\sum_{i-1}^{n} a_i \leq x + y \leq \sum_{i=1}^{n} a_i + G$. If such a point (x,y) exists, the corresponding solution is optimal. If not, go to Step 7.

7. For each task in Set III such that $b_i \geq G + 2$,

   7A. Form Set III' by deleting $a_i' b_i' c_i'$ from Set III.

   7B. Repeat Steps 3 through 5 for Set III' instead of III.

   7C. Select the solution corresponding to the point (x,y) where

   $$\sum_{i=1}^{n} a_i - (b_i' - (G-1)) \leq x + y \leq \sum_{i=1}^{n} a_i - 1$$

   If such a point (x,y) exists, the optimal solution is of

the second form with $S = a_i'b_i'c_i'$. Stop. If such a point (x,y) does not exist, continue with Step 7 iteration.

8. No solution is possible.

Theorem 4.3. Algorithm 4.3 provides an optimal solution to Problem 4.5, if one exists.

Proof.

Assume a feasible optimal solution exists, but Algorithm 4.5 fails. Failure occurs only at Step 8. Neither of the two solution forms could be found. Johnson's theorem requires that a specific order of tasks forms a feasible solution in Set R and Set T, if a solution exists. The pre-ordering of tasks in Set III in Steps 1 and 2 of Algorithm 4.3 arranges the tasks in the order they become candidates to be assigned in Set R and Set T. Based on the definitions of Step 3 and the initial point in Step 4, Step 5 decides whether a task may be assigned in Set R, in Set T, or in both sets. For each decision, a new point is reached in the graph. After each task in Set III has been tried, Step 6 and Step 7C find if any point in the graph represents a feasible solution. The feasible solution is that solution which satisfies one of the two basic forms. The points on the graph always represent feasible solutions because others are deleted at Step 5B. Since all attempts are made to satisfy one of the two basic forms, the algorithm's failure means no solution. A contradiction.

4.2.5. Extension to the Four Stage Problem

In this section we discuss a four stage problem which parallels the Special Segmented Problem of an earlier section. Although the

extension from the three stage problem is straightforward, Problem 4.6 formally describes the new situation.

Problem 4.6.

Find a schedule for n tasks composed of four operations. The first and third operations must be performed on Machine One, and the second and fourth operations must be performed on Machine Two ($m = 2$). Using Jackson's notation of Section 2.2 the tasks are divided into two sets: {ABAB} and {BABA}. The form of the solution is restricted as described below to a schedule with no idle time.

Machine One: The first operations of the set {ABAB}, followed by the second operations of the set {BABA}, followed by the third operations of the set {ABAB}, and followed by the fourth operations of the set {BABA}.

Machine Two: The first operations of the set {BABA}, followed by the second operations of the set {ABAB}, followed by the third operations of the set {BABA} and followed by the fourth operations of the set {ABAB}.

If the tasks in Problem 4.6 are limited to those whose first and fourth operations are equal and whose second and third operations are equal, the knapsack solution of Problem 4.5 is again applicable. From the Johnson result the processing order of the tasks is the same on each processor. Since no contribution is made by the second and third operations, all second operations must be feasible initially from the end of stage 3. The extra operation, therefore, does not make the problem different from Problem 4.5. Algorithm 4.3 may be applied.

4.5 Other Subproblems

The results presented in this chapter strongly restrict the size of the operations of the tasks we consider. The operations must differ in length by a constant or pairs of operations must have identical length. The results do not apply for tasks with operations of arbitrary lengths. We, however, have expanded the list of solved subproblems. These solutions may be incorporated into solutions of more complex problems.

New problems may have solutions which apply several ideas we described earlier. For example, one attack on new problems may be to find a canonical form for the m, $m \geq 4$, segment problem and to apply the knapsack solution method to portions of the problem. Significant use also may be made of the decoupling phenomenon in conjunction with these canonical forms. In any event, it appears likely that research in quest of efficient algorithms for scheduling subproblems will go on long into the future.

Chapter 5

The Four Processor Problem

5.1. The Four Processor Problem

Szwarc's work [1968] was described in Section 2.3. He considered a three processor problem and tasks consisting of a chain of three operations. In this section the results are extensions of Szwarc's work to the $4 \times n$ sequencing problem. In Section 5.2 we define the problem so that we are seeking the minimal completion time for all tasks on all machines. Then Section 5.3 develops a condition on the operations so that the order of tasks is identical on adjacent processors and the total completion time is minimal. Section 5.4 follows Szwarc's objective and reduces the $4 \times n$ sequencing problem. Under two explicit conditions the $4 \times n$ sequencing problem is reduced to n $2 \times n$ sequencing problems and a $3 \times n$ sequencing problem, respectively. (Theorem 5.2 and Theorem 5.3)

5.2. Problem Definition

Before discussing the new results we first present the definition of the four processor problem and the notation we use throughout the chapter.

Problem 5.1. The Four Processor Problem

Each of n tasks is composed of four operations a_i, b_i, c_i, and d_i to be executed on four processors, A, B, C, and D, respectively ($m = 4$). Find the schedule which minimizes the total completion time.

From Johnson's result in Theorem 2.3 we know that two permutations

of the tasks are sufficient to assign the n tasks to the four processors. Let us call the permutations p and q where

$$p = (p1, p2, \ldots , pi, pi+1, \ldots , pn)$$

$$q = (q1, q2, \ldots , qi, qi+1, \ldots , qn)$$

Each pj and qj, j = 1, 2, ... , n, represents one of the n tasks. In general, pj $\neq$ qj for j = 1, 2, ... , n. The permutation p represents the permutation of tasks on the first two processors, A and B; permutation q represents the permutation of tasks on the last two processors, C and D. A schedule is respectively $A_p B_p C_q D_q$. For example, if p = (1,3,2) and q = (2,1,3) the schedule represented by $A_p B_p C_q D_q$ on processors A, B, C, and D would be

A: a_1 a_3 a_2
B: b_1 b_3 b_2
C: c_2 c_1 c_3
D: d_2 d_1 d_3

5.3. Restrictions on Permutations p and q

In some cases it is possible to consider four processor schedules in which the order of tasks on each processor is the same. That is, p equals q. The result of Theorem 5.1, however, is more general. This theorem describes a case when two adjacent processors in the $m \times n$ sequencing problem may process tasks in identical order without loss of optimality.

<u>Theorem 5.1</u>. If $\max_k t_k \leq \min_k s_k$ where the operation s_k on machine S immediately precedes the operation t_k on machine T in the $m \times n$ sequencing problem, then one optimal schedule is one in which the permutation p of operations on machine S is identical to the

permutation q of operations on machine T.

Proof.

$p = (p1, p2, \ldots, pn)$ is the permutation of operations on machine S and $q = (q1, q2, \ldots, qn)$ is the permutation of operations on machine T in an optimal schedule. Assume for some minimal i, $pi \neq qi$. Processor T is idle while s_{pi+1} is executed on processor S. Since $s_{pi+1} \geq t_{pi}$, by assumption, t_{pi} may be executed on processor T while s_{pi+1} is executed. No task on processor T completes later in this new schedule than it did in the original feasible schedule. Hence, the new schedule is also feasible. Since this process may be repeated until $p = q$, one optimal schedule is one in which $p = q$.

Corollary 5.1. In Problem 5.1 if $\max_k c_k \leq \min_k b_k$, the permutation of operations on each processor is identical.

Proof.

By Johnson's result which was restated in Theorem 2.3 the permutation of operations is identical on processors A and B and is also identical on processors C and D. By Theorem 5.1 and the hypothesis the permutation of operations is identical on processors B and C. Hence, the permutation of operations is identical on all processors.

5.4. Extension of Szwarc's Results

The $3 \times n$ sequencing problem results in Section 3.2 were by Wlodzimierz Szwarc, and T. S. Arthanari and A. C. Mukhopadhyay. The $4 \times n$ sequencing problem results here use a similar formulation. In all cases the total idle time on the last processor is to be minimized.

To discuss this work on the four processor problem we must

consider three quantities. Assuming the two permutations p and q are

$$p = (p1, p2, \dots, pn)$$

$$q = (q1, q2, \dots, qn) ,$$

we may examine the values of

x_k -- idle time on the second processor after assigning task pk

y_k -- idle time on the third processor after assigning task qk

z_k -- idle time on the fourth processor after assigning task qk

After assigning tasks p1 and q1 to the respective four machines,

$$x_1 = a_{p1}$$

$$y_1 = \max\left(b_{p1} + x_1 , b_{q1} + R_{q1}\right)$$

$$z_1 = y_1 + c_{q1}$$

where $R_{qj} = \sum_{i=1}^{\ell} x_i + \sum_{i=1}^{\ell-1} b_{pi}$ when $p\ell = qj$.

Consequently, $R_p = \sum_{i=1}^{\ell} x_i + \sum_{i=1}^{\ell-1} b_{pi}$

This term represents the possibility that p and q are not identical. Then tasks on the third processor may not begin execution immediately after the first task is completed on the second processor.

After assigning tasks p2 and q2 to the respective four machines,

$$x_2 = \max\left(a_{p1} + a_{p2} - x_1 - b_{p1} , 0\right)$$

$$y_2 = \max\left(R_{q2} + b_{p2} - (y_1 + c_{q1}) , R_{p2} + b_{p2} - (y_1 + c_{q1}) , 0\right)$$

$$z_2 = \max\left(y_1 + y_2 + c_{q1} + c_{q2} - z_1 - d_{q1} , 0\right)$$

In general, after assigning tasks pk and qk, $k = 1, 2, \dots, n$, to the respective four machines the idle times are:

$$x_k = \max\left(\sum_{i=1}^{k} a_{pi} - \sum_{i=1}^{k-1} x_i - \sum_{i=1}^{k-1} b_{pi} , 0\right)$$

$$y_k = \max\left(R_{qk} + b_{pk} - \sum_{i=1}^{k-1} y_i - \sum_{i=1}^{k-1} c_{qi} ,\right.$$

$$R_{pk} + b_{pk} - \sum_{i=1}^{k-1} y_i - \sum_{i=1}^{k-1} c_{qi} , 0)$$

$$z_k = \max \left(\sum_{i=1}^{k} y_i + \sum_{i=1}^{k} c_{qi} - \sum_{i=1}^{k-1} z_i - \sum_{i=1}^{k-1} d_{qi} , 0 \right)$$

However, what we must minimize is the total idle time on the fourth processor. The total idle time on the fourth processor is Z_n.

$$Z_n = \sum_{i=1}^{n} z_i = \max \left(\sum_{i=1}^{n} y_i + \sum_{i=1}^{n} c_{qi} - \sum_{i=1}^{n-1} d_{qi} , \sum_{i=1}^{n-1} z_i \right)$$

Similarly, we may obtain the total idle time on the second processor, X_n, and the total idle time on the third processor, Y_n.

$$X_n = \sum_{i=1}^{n} x_i = \max \left(\sum_{i=1}^{n} a_{pi} - \sum_{i=1}^{n} b_{pi} , \sum_{i=1}^{n-1} x_i \right)$$

$$Y_n = \sum_{i=1}^{n} y_i = \max \left(R_{qn} + b_{pn} - \sum_{i=1}^{n-1} c_{qi} , \; R_{pn} + b_{pn} - \sum_{i=1}^{n-1} c_{qi} , \sum_{i=1}^{n-1} y_i \right)$$

To simplify the expression for X_n we define the quantity K_u.

$$K_u = \sum_{i=1}^{u} a_{pi} - \sum_{i=1}^{u-1} b_{pi}$$

$$X_u = \sum_{i=1}^{u} x_i$$

$$X_n = \max (K_n , K_{n-1}) = \max_{1 \le u \le n} K_u$$

Similarly to simplify Y_n we first expand the expression for Y_n which becomes

$$Y_n = \max \left(\sum_{i=1}^{\ell} x_i + \sum_{i=1}^{\ell-1} b_{pi} - \sum_{i=1}^{n-1} c_{qi} , \; \sum_{i=1}^{n} x_i + \sum_{i=1}^{n} b_{pi} - \sum_{i=1}^{n-1} c_{qi}, \sum_{i=1}^{n-1} y_i \right)$$

where $p\ell = qn$

$$Y_n = \max \Big(\sum_{i=1}^{n} x_i + \sum_{i=\ell+1}^{n} b_{pi} - \sum_{i=n+1}^{\ell} x_i + \sum_{i=1}^{n} b_{pi} - \sum_{i=\ell}^{n} b_{pi} + \sum_{i=n}^{\ell} b_{pi} - \sum_{i=1}^{n-1} c_{qi} ,\ \sum_{i=1}^{n} x_i + \sum_{i=1}^{n} b_{pi} - \sum_{i=1}^{n-1} c_{qi} ,\ \sum_{i=1}^{n-1} y_i \Big)$$

where $p\ell = qn$.

Now H_v and G_v are defined for substitution in Y_n.

$$H_v = \sum_{i=1}^{v} b_{pi} - \sum_{i=1}^{v-1} c_{qi}$$

$$G_v = - \sum_{i=\ell+1}^{n} x_i + \sum_{i=v+1}^{\ell} x_i - \sum_{i=\ell}^{v} b_{pi} + \sum_{i=v}^{\ell} b_{pi} \quad \text{where } p\ell = qv.$$

$$Y_n = \max \Big(G_n + H_n + \max_{1 \le u \le n} R_u ,\ H_{n-1} + \max_{1 \le u \le n} R_u ,\ G_{n-1} + H_{n-1} + \max_{1 \le u \le n-1} R_u , H_{n-1} + \max_{1 \le u \le n-1} R_u ,\ \ldots ,\ G_1 + H_1 + R_1 \Big)$$

$$Y_n = \max \Big(H_n + \max_{1 \le u \le n} R_u + \max (G_n , 0) ,\ H_{n-1} + \max_{1 \le u \le n} R_u + \max (G_{n-1} , 0),\ \ldots ,\ H_1 + R_1 + \max (G_1 , 0) \Big)$$

$$Y_n = \max_{1 \le u \le v \le n} \big(H_v + K_u + \max (G_v , 0) \big)$$

By defining F_w, Z_n may likewise be simplified.

$$F_w = \sum_{i=1}^{w} c_{qi} - \sum_{i=1}^{w-1} d_{qi}$$

$$Z_n = \max \Big(\max_{1 \le u \le v \le n} (H_v + R_u + \max (G_v , 0)) + F_n ,$$

$$\max_{1 \le u \le v \le n-1} (H_v + R_u + \max (G_v, 0)) + F_{n-1} ,$$

$$\ldots ,$$

$$H_1 + R_1 + \max (G_1 , 0) + F_1 \Big) \qquad \text{Equation 5.1}$$

In order to make it obvious that Z_n , the idle time on processor D, depends on the permutations p and q, we rename Z_n to be $g(p,q)$.

From these equations and following Arthanari and Mukhopadhyay, two results for the four machine problem arise in Theorems 5.2 and 5.3. Figure 5.1 shows an example of the use of Theorem 5.2 while Figure 5.2. shows an example of Theorem 5.3.

Theorem 5.2. If n tasks, each composed of four operations, a_i, b_i, c_i, and d_i, are to be executed, respectively, on four processors ($m = 4$) and $\max_k b_k \le \min_k c_k$ and $\max_k a_k \le \min_k b_k$, the four machine problem is solved by solving n two-machine problems.

Proof.

Let the permutations of tasks on the first two processors be $\bar{p}$ and on the last two processors be $\bar{q}$. For each t, $1 \le t \le n-1$, since $\max_k b_k \le \min_k c_k$,

$$\bar{H}_t = \sum_{i=1}^{t} b_{\bar{p}i} - \sum_{i=1}^{t-1} c_{\bar{q}i}$$

$$\bar{H}_{t+1} = \sum_{i=1}^{t+1} b_{\bar{p}i} - \sum_{i=1}^{t} c_{\bar{q}i}$$

$$\bar{H}_t - \bar{H}_{t+1} = - b_{\bar{p}t+1} + c_{\bar{q}t} \ge 0$$

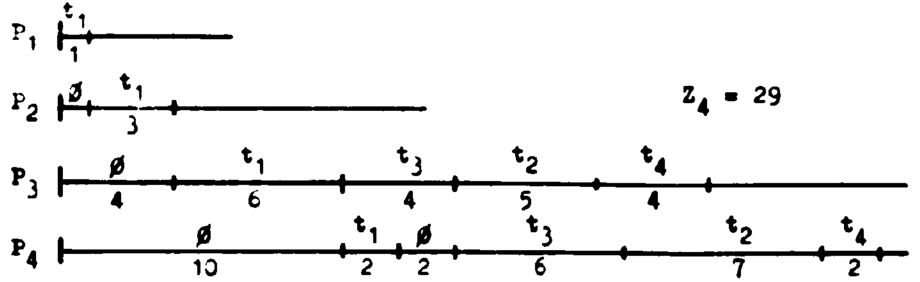
$$\bar{H}_t \ge \bar{H}_{t+1}$$

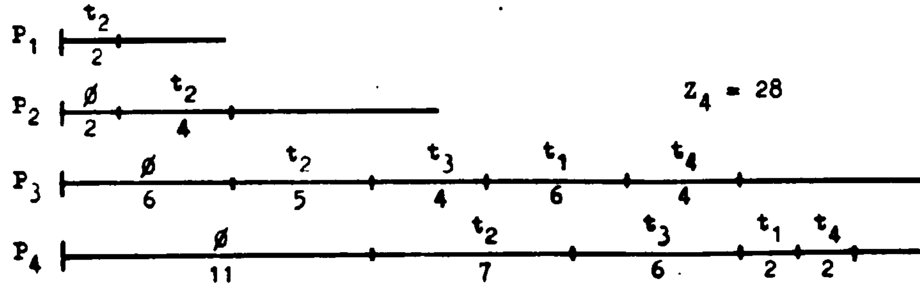
Tasks

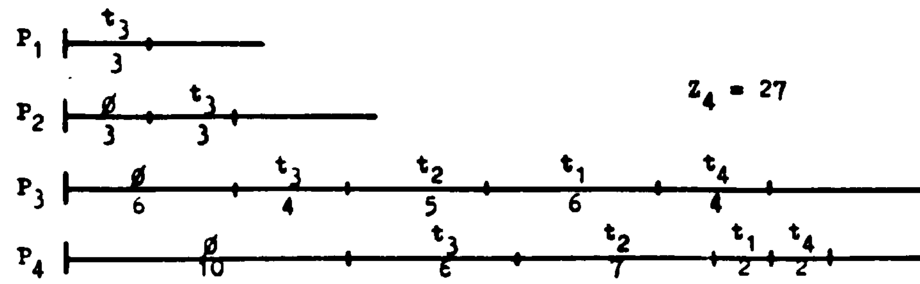
$t_1 = (1,3,6,2)$
$t_2 = (2,4,5,7)$
$t_3 = (3,3,4,6)$
$t_4 = (1,4,4,2)$

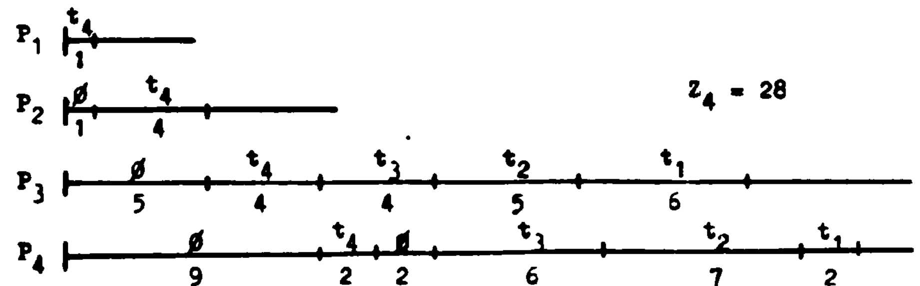
Figure 5.1. Example of Theorem 5.1

Four possibilities of a solution









Optimal permutation is (3,2,1,4)

Tasks

$t_1 = (2,4,3,6)$

$t_2 = (3,6,1,4)$

$t_3 = (4.5,3,2)$

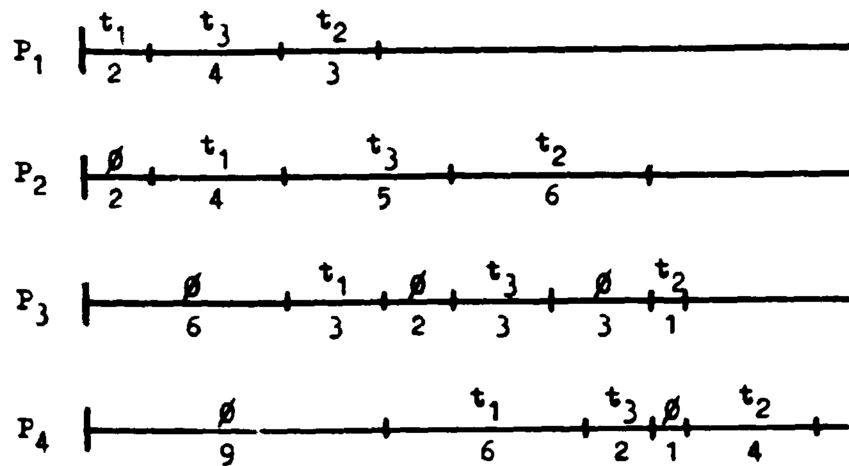


Figure 5.2. Example of Theorem 5.2

Therefore,

$$\max_{1 \le t \le n} \bar{H}_t = \bar{H}_1$$

Similarly, for each t, $1 \le t \le n-1$, since $\max_k a_k \le \min_k b_k$,

$$\bar{K}_t = \sum_{i=1}^{t} a_{\bar{p}i} - \sum_{i=1}^{t-1} b_{\bar{p}i}$$

$$\bar{K}_{t+1} = \sum_{i=1}^{t+1} a_{\bar{p}i} - \sum_{i=1}^{t} b_{\bar{p}i}$$

$$\bar{K}_t - \bar{K}_{t+1} = - a_{\bar{p}t+1} + b_{\bar{p}t} \ge 0$$

$$\bar{K}_t \ge \bar{K}_{t+1}$$

Therefore,

$$\max_{1 \le t \le n} \bar{K}_t = \bar{K}_1$$

By Theorem 5.1 and Corollary 5.1 and the fact that

$\max_k b_k \le \min_k c_k$ and $\max_k a_k \le \min_k b_k$, the permutation $\bar{p}$ is identical to

the permutation $\bar{q}$. Then

$$\bar{G}_v = 0 \text{ for } v = 1, 2, \ldots, n$$

Using equation 5.1 above

$$g(\bar{q},\bar{q}) = \max_{1 \le w \le n} (\bar{F}_w + \bar{H}_1 + \bar{K}_1) =$$

$$= \bar{H}_1 + \bar{K}_1 + \max_{1 \le w \le n} (\bar{F}_w)$$

$$= H_1 + K_1 + \max_{1 \le w \le n} (c_{\bar{q}1}, c_{\bar{q}1} + c_{\bar{q}2} - d_{\bar{q}1}, \bar{F}_3, \ldots, \bar{F}_w)$$

$$= H_1 + K_1 + c_{\bar{q}1} - d_{\bar{q}1} + \max[d_{q1}, \max_{2 \le w \le n} \sum_{i=2}^{w} c_{\bar{q}1} - \sum_{i=2}^{w-1} d_{\bar{q}1})]$$

Equation 5.2

Our goal is to minimize $g(\bar{q},\bar{q})$ over all possible sequences $\bar{q}$ designated by the set Q_i. For a fixed $i = \bar{q}1$,

$$I_i = \min_{\bar{q} \in Q_i} (\max_{2 \le w \le n} (\sum_{i=2}^{w} c_{\bar{q}i} - \sum_{i=2}^{w-1} d_{\bar{q}i}))$$

I_i is the minimum idle time on the last machine D from the two machine problem with machines C and D where the set of tasks is all n tasks except task $i = \bar{q}1$. Call this optimal permutation S_i. Letting D_i and $g_i(\bar{q},\bar{q})$ be the idle time on machine D when task i is task $\bar{q}1$, from Equation 5.2

$$D_i = g_i(\bar{q},\bar{q}) = a_{\bar{q}1} + b_{\bar{q}1} + c_{\bar{q}1} - d_{\bar{q}1} + \max\,(c_{\bar{q}1},\ I_{\bar{q}1}).$$

The problem thus is to find i_o and an optimal permutations S_{i_o} such that $D_{i_o} = \min_{1 \le i \le n} g_i(\bar{q},\bar{q})$. The complete optimal permutation is $(i_o,\ S_{i_o})$.

<u>Theorem 5.3</u>. If n tasks each composed of four operations a_i, b_i, c_i, and d_i are to be executed, respectively, on four processors A, B, C and D ($m = 4$), and $\max_k d_k \le \min_k c_k$, then the four machine problem reduces to a three machine problem in which permutation p is not necessarily the same as permutation q.

Proof.

Since $\max_k d_k \le \min_k c_k$, for each t, $1 \le t \le n-1$,

$$\bar{F}_t = \sum_{i=1}^{t} c_{\bar{q}1} - \sum_{i=1}^{t-1} d_{\bar{q}1}$$

$$\bar{F}_{t+1} = \sum_{i=1}^{t+1} c_{\bar{q}1} - \sum_{i=1}^{t} d_{\bar{q}1}$$

$$\bar{F}_t - \bar{F}_{t+1} = -\,c_{\bar{q}t+1} + d_{\bar{q}t} \le 0$$

$$\bar{F}_{t+1} \ge \bar{F}_t$$

Therefore,

$$\max_{1 \le w \le n} \bar{F}_w = \bar{F}_n$$

From Equation 4.2 and definitions of H_v, K_u, and F_w here and in Theorem 5.2,

$$g(\bar{p},\bar{q}) = \max_{1 \le u \le v \le w \le n} (\bar{F}_w + \bar{H}_v + \max(\bar{G}_v, 0) + \bar{K}_u)$$

$$= \max_{1 \le w \le n} (\bar{F}_w + \max_{1 \le u \le v \le w} (\bar{H}_v + \max(\bar{G}_v, 0) + \bar{K}_u))$$

Let $\mathcal{J}(p,q)$ be the time for a complete schedule using permutation p and q. Then

$$\mathcal{J}(\bar{p},\bar{q}) = g(\bar{p},\bar{q}) + \sum_{i=1}^{n} d_{\bar{q}i}$$

$$\mathcal{J}_{min}(\bar{p},\bar{q}) = \min_{\bar{p} \in P_i,\ \bar{q} \in Q_i} \mathcal{J}(\bar{p},\bar{q})$$

$$= \min_{\bar{p} \in P_i,\ \bar{q} \in Q_i} [\ g(\bar{p},\bar{q}) + \sum_{i=1}^{n} d_{\bar{q}i}\ ]$$

$$= \min_{\bar{p} \in P_i,\ \bar{q} \in Q_i} [\ \sum_{i=1}^{n} c_{\bar{q}i} - \sum_{i=1}^{n-1} d_{\bar{q}i} + \sum_{i=1}^{n} d_{\bar{q}i} + \max_{1 \le u \le v \le n} (\bar{H}_v + \max(\bar{G}_v, 0) + \bar{K}_u)\ ]$$

$$= \sum_{i=1}^{n} c_i + \min_{\bar{p} \in P_i,\ \bar{q} \in Q_i} [\ d_{\bar{q}n} + \max_{1 \le u \le v \le n} (\bar{H}_v + \max(\bar{G}_v, 0) + \bar{K}_u)\ ]$$

From the above the idle time on processor C is exactly

$$\max_{1 \le u \le v \le n} (\bar{H}_v + \max(\bar{G}_v, 0) + \bar{K}_u)$$

Hence, $\mathcal{J}_{min}(\bar{p},\bar{q})$ is found by finding the solution to the three processor problem of machines A, B, and C with $\bar{p} \neq \bar{q}$ in general.

5.5. In Summary

As we pointed out several times in earlier chapters the significance of results for simplified subproblems is that they may later be incorporated in the solutions of larger problems. Our results and those of Szwarc demonstrate this significance most vividly. We have taken the more complex $4 \times n$ sequencing problem and shown that in certain

instances only less complex problems need to be solved.

However, the work in this area is far from complete. The general four processor problem is not solved efficiently. In addition, little reduction has been made to the m, $m > 4$, processor problem. Future researchers may, nevertheless, use the same methods in the more complicated problems.

Chapter 6

Future Directions

The results in $m \times n$ sequencing research has had three fruitful periods, the mid 1950,s, the early 1960's, and the early 1970's. The problems are noted for their simple formulation and the elusiveness of efficient algorithms.

We believe that one of the most important contributions of the earlier results and those presented here is to the understanding and analysis of more complex problems. The realistic problems of computer scheduling bear only minimal resemblance to the problems presented. But before the more complex scheduling problems may be handled satisfactorily, we must know the fundamental results of $m \times n$ sequencing.

To predict the future successes in this area is a risky, if enjoyable, job. Yet several problems seem ripe for solution in the near future. First, the recent Coffman and Graham results lend hope that the sequencing of 1-unit tasks with tree precedence or perhaps acyclic precedence on more than two processors may have an efficient solution. Likewise, the work here gives hope for an efficient, complete solution of the sequencing problem with 1-unit and 2-unit tasks with acyclic precedence.

Second, the Special Segmented Problem is closely related to the three processor problem. It may be possible also to find significant solutions to this set of problems. The concept of decoupling between sets of tasks may be a powerful key to these efficient solutions.

No computer scientist, however, may ignore the work in the field of computational complexity. Although no problem with an inherently

exponential solution is known, such problems may be found. Although disappointing, these results would be useful in bounding the area for future researchers.

BIBLIOGRAPHY

[Arthanari 1971] Arthanari, T. S. and Mukhopadhyay, A. C., "A Note on a Paper by W. Szwarc," Naval Research Logistics Quarterly, 18, No. 1, March 1971, pp. 135-138.

[Bauer 1970] Bauer, H. and Stone, H., "The Scheduling of N Tasks with M Operations on Two Processors," Report No. STAN-CS-70-165, Computer Science Department, Stanford University, July 1970.

[Bellman 1956] Bellman, R., "Mathematical Aspects of Scheduling Theory," Journal of SIAM, 4, No. 3, September 1956, pp. 168-185.

[Chandy 1972a] Chandy, K. M., Dickson, J. R., and Ramamoorthy, C. V., "Optimal Real-Time Basic Scheduling of Two Processors," Fifth Annual Hawaii International Conference on Systems Sciences, January 1972, pp. 216-218.

[Chandy 1972b] Chandy, K. M., Dickson, J. R., and Ramamoorthy, C. V., "Optimal Scheduling Disciplines for Two-Processor Systems," Research Report, Department of Computer Sciences, University of Texas at Austin, 1972.

[Clark 1947] Clark, W., The Gantt Chart. London: Pitman and Sons, 1947.

[Coffman 1972] Coffman, E. G., Jr., and Graham, R. L., "Optimal Scheduling for Two-Processor Systems," ACTA Informatica, 1, No. 3, February 1972, pp. 200-213.

[Conway 1967] Conway, R. W., Maxwell, W. L., and Miller, L. W., Theory of Scheduling. Reading, Massachusetts: Addison-Wesley Publishing Company, 1967.

[Day 1970] Day, J. E. and Hottenstein, M P., "Review of Sequencing Research," Naval Research Logistics Quarterly, 17, No. 1, March 1970, pp. 11-39.

[Dudek 1964] Dudek, R. A. and Teuton, O. F., Jr., "Development of M-Stage Decision Rule for Scheduling n Jobs through M Machines," "Operations Research, 12, No. 3, May 1964, pp. 471-497.

[Hsu 1966] Hsu, N. C., "Elementary Proof of Hu's Theorem on Isotone Mappings," Proceedings of American Mathematical Society, February 1966, pp. 111-114.

[Hu 1961] Hu, T. C., "Parallel Sequencing and Assembly Line Problems," Operations Research, 9, No. 6, November 1961, pp. 841-848.

[Jackson 1956] Jackson, J. R., "An Extension of Johnson's Result on Job-Lot Scheduling," Naval Research Logistics Quarterly, 3, No. 3, September 1956, pp. 201-203.

[Johnson 1954] Johnson, S. M., "Optimal Two- and Three-Stage Production Schedules with Setup Times Included," Naval Research Logistics Quarterly, 1, No. 1, March 1954, pp. 61-68.

[Karush 1965] Karush, W., "A Counterexample to a Proposed Algorithm for Optimal Sequencing of Jobs," Operations Research, 13, No. 2, March 1965, pp. 323-325.

[Smith 1966] Smith, R. D. and Dudek, R. A., "A General Algorithm for Solution of the n-Job, M-Machine Sequencing Problem of the Flow Shop," Operations Research, 15, No. 1, December 1966, pp. 71-82.

[Smith 1969] Smith, R. D. and Dudek, R. A., "Errata," Operations Research, 17, No. 4, July 1969, p. 759.

[Szwarc 1968] Szwarc, W., "On Some Sequencing Problems," Naval Research Logistics Quarterly, 15, No. 2, June 1968, pp. 127-155.