Division of Research
Graduate School of Business Administration
The University of Michigan

December 1982

An Integer Programming Algorithm With Network Cuts for Solving the Assembly Line Balancing Problem

Working Paper No. 321

F. Brian Talbot
The University of Michigan

James H. Patterson
The University of Missouri-Columbia

FOR DISCUSSION PURPOSES ONLY

None of this material is to be quoted or reproduced without the express permission of the Division of Research.

Abstract

In this paper, we describe an integer programming algorithm for assigning tasks on an assembly line to work stations in such a way that the number of work stations is minimal for the rate of production desired. The procedure insures that no task is assigned to a work station before all tasks which technologically must be performed before it have been assigned (precedence restrictions are not violated), and that the total time required at each work station performing the tasks assigned to it does not exceed the time available (cycle time restrictions are not violated). The procedure is based on a systematic evaluation (enumeration) of all possible task assignments to work stations. Significant portions of the enumeration process are performed implicitly, however, by utilizing tests described in the paper which are based on the specific structure of the line balancing problem. An artifice termed a network cut is also developed which eliminates from explicit consideration the assignment of tasks to work stations where such assignments would not lead to improved line balances. Results reported demonstrate that the procedure can obtain optimal balances for assembly lines with between fifty and one-hundred tasks in a reasonable amount of computation time and with modest computer storage requirements.

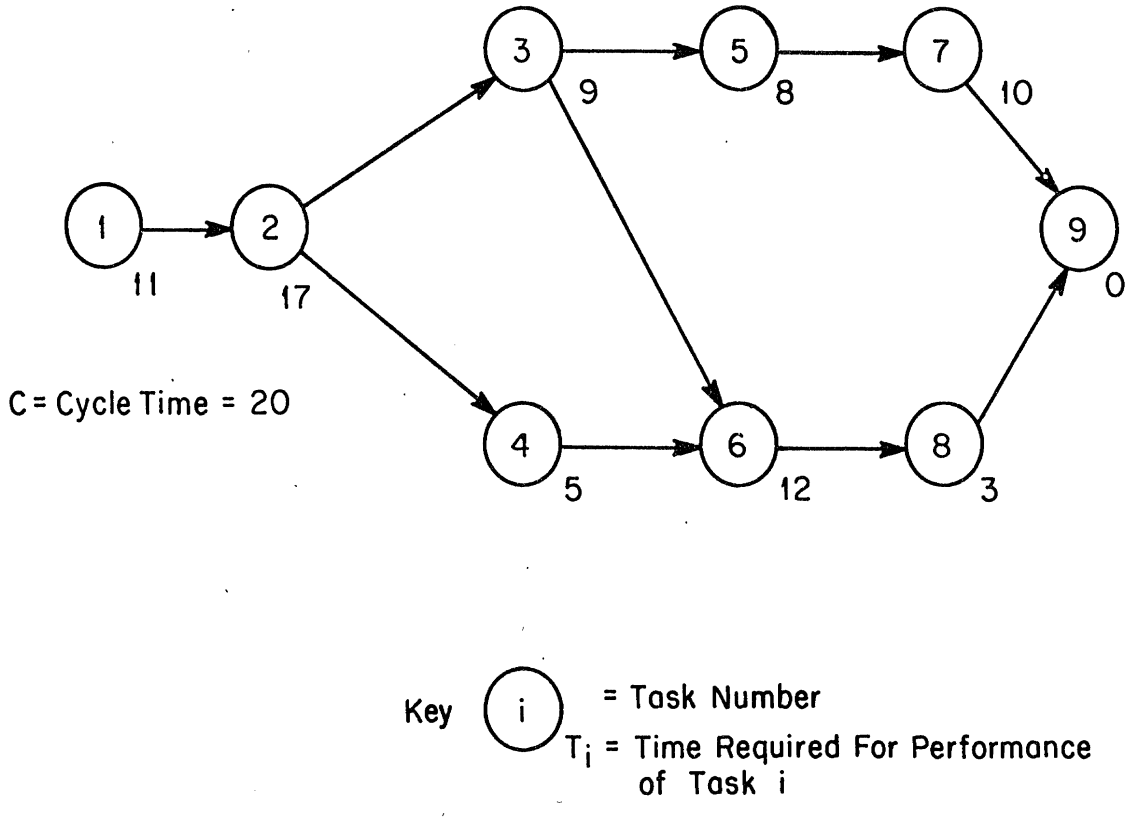
1. Introduction

An integer programming algorithm is presented for determining the minimum number of work stations required to balance a paced assembly line, subject to constraints on the sequence in which tasks may be performed, and subject to a limit (the cycle time) on the time which can be expended in performing tasks at each work station. The cycle time in these problems is based on a production plan or an output rate specified to meet forecasted demand. If the demand is 120 units per hour, for example, then the cycle time is thirty seconds, or one-half minute per work station. A related balancing problem, which is not considered in this paper, involves the determination of a minimal cycle time (or maximum output rate) for a fixed line length or number of work stations. In this problem, precedence constraints must not be violated, and the given number of work stations must not be exceeded in determining the minimum cycle time balance. Heuristic techniques for solving the first problem can be generalized to solve the second by successively increasing the cycle time until a balance is achieved for the stipulated number of work stations. Our procedure can also be applied in an analogous fashion to solve this latter problem.

Indicative of the type of problem encountered in balancing an assembly line is the one given in Figure I, adapted from [3]. In this network representation of the assembly line balancing (ALB) problem, nodes indicate tasks to be performed, and arcs denote the order in which the tasks must be performed. For example, in Figure I, task six cannot be begun until tasks three and four have both been completed. Associated with each task is the time required for its performance, T_i. The cycle time C gives the maximum amount of time allowed at a work station to perform the tasks assigned to it. We will assume in the development of the procedure that the nodes of the network are numbered such

Figure I

Precedence Diagram of Bowman's [3] Problem
with Dummy Terminal Node Appended



that if task m precedes task n, $m < n$. Furthermore, we will assume that a unique terminal dummy task with label N and time zero is appended to the network as shown in Figure I (task 9). We assume that this is the only task with zero time. Such node numbering schemes in a network are not unique, and in fact determine the order in which tasks are considered for assignment to a work station.

As is the case with many of the recent attempts to solve this problem with the use of integer programming, our approach is based on the Balas implicit enumeration algorithm [2]. Our procedure differs from these other approaches, however, in that we use integer variables rather than 0-1 (binary) variables. This results in a significant reduction in computer storage requirements vis-à-vis these other methods. Other optimization approaches for solving this problem are described in [5, 6, 8, and 13].

Notation which will be used to describe the procedure is given in Table I. In Section 2 of the paper, we state the integer programming formulation of this problem, and contrast it to the existing binary programming formulations. In Section 3 the basic solution procedure is presented, and in Section 4 we describe several techniques for exploiting problem structure and accelerating the convergence of our approach. Computational experience in solving a series of test problems found in the open literature is found in Section 5. Concluding remarks are given in Section 6.

Table I

Definition of Terms

Symbol (Listed Alphabetically)	Definition
A_i	A nonnegative integer variable which is equal to the number of the work station in which task i is assigned. (A_i=0 indicates that task i has not been assigned to a work station.)
B_i	The station assignment for task i in the current best solution. Initially, B_N is set equal to H.
C	Cycle time.
H	The number of stations in a heuristic solution to the ALB problem. (H equals N if a heuristic solution is not known.)
I_j	Idle time at station j: i.e., C minus the sum of the task times for tasks currently assigned to station j.
i	Task number: $(1 \leq i \leq N)$.
j	Station number: $(1 \leq j \leq H)$.
M	A lower bound on the number of work stations required.
N	Number of tasks to be assigned. (Also equal to the identifying number of the unique dummy terminal task without successors.)
P_i (S_i)	The set of tasks which must precede (succeed) task i.
P_i^* (S_i^*)	The set of tasks which must immediately precede (succeed) task i.
T_i	Time required to perform task i.
T*	Maximum task time in a given problem. $T^* = \max_i \{T_i\}$.
U_i (L_i)	An upper (lower) bound specifying the highest (lowest) numbered station into which task i may be assigned based on cycle time and precedence restrictions.
W_j	The set of all tasks which can be assigned to station j by virtue of task precedence constraints.
W'_j	The subset of tasks in W_j that are actually assigned to station j.

2. Integer Programming Formulation

Conceptually, the assembly line balancing (ALB) problem is formulated as follows:

Minimize: A_N

Subject to: $\sum_{i \in W'_j} T_i \leq C$ for $j=1,\ldots,H$ (1)

$A_m \leq A_n$ $m \in P_n^*$

for $n=1,\ldots,N$ (2)

The objective is to minimize the number of work stations, or equivalently, A_N, which is the station to which the unique terminal task N is assigned. Constraints (1) insure that the sum of the task times for tasks assigned to a particular station does not exceed the cycle time, C. Constraints (2) insure that all technological predecessors of a given task are performed before it is performed.

The primary obstacle to implementing this formulation directly is the difficulty in operationally specifying (1). Prior integer programming attempts [3, 15, 20, 22] have led to formulations using 0-1 variables, which do permit the explicit representation of cycle, occurrence, and precedence constraints. The consequence of using 0-1 variables, however, is that the resulting formulation contains a large number of variables and constraints. This is demonstrated in [15], where four 0-1 formulations of the ALB problem are compared by means of an eight-task example problem. The approach involving the fewest number of variables and constraints requires 29 variables and 23 constraints for the given eight-task problem. For much larger industrial size problems--say, 50 to 100 tasks--this conservatively translates into formulations involving literally hundreds of variables and constraints. Furthermore, if the

search procedure for problem solution works "down" from a heuristic line length, H, both the number of variables and the number of constraints needed in these formulations are a function of the best known heuristic solution. The larger H is relative to the lower bound number of work stations, M, the larger is the resulting formulation.

The proposed solution method eliminates the need to explicitly formulate (1) and (2), which significantly reduces the computer storage requirements relative to prior integer programming approaches. The number of variables, A_i, is simply equal to the number of assembly tasks, N. Constraints are not explicitly formulated at all in our procedure. Precedence relationships are maintained by directly testing an immediate predecessor array, P_i^*. Furthermore, cycle time restrictions are invoked through the evaluation of an idle time vector, I. Unique task assignment to a work station is insured by assigning tasks in numerical order, starting with task one and terminating with task N. Thus, the basic proposed formulation requires computer storage of only four vectors, each of dimension no greater than 1xN, and one NxK matrix, where K is the number of immediate predecessors for the task with the maximum number of immediate predecessors. The vectors are A, I, T, and U, and the matrix is P*. Depending upon the type of heuristic rules employed, a few additional 1xN vectors may also be required.

3. The Solution Procedure

The basic solution methodology will now be described. First, upper (U_i) and lower (L_i) bounds on the stations into which a task may be assigned are determined. The problem is then solved by using our modified Balas algorithm, which guarantees that an optimal solution will be found.

The Determination of Upper and Lower Bounds on Task Assignment

In order to restrict the enumeration of possible work task assignments, an upper bound on the latest feasible station into which a task can be assigned is determined as follows:

$$U_i = H - [(T_i + \sum_{k \epsilon S_i} T_k) / C]^+ \qquad \text{for } i=1,\ldots,N-1 \qquad (3)$$

and, $U_N = H-1$

where $[x]^+$ denotes the smallest integer $\geq x$. This states that the upper bound on the terminal task is one station less than the current best known solution H.

Analogously, (4) identifies the earliest possible station to which a task can be assigned by virtue of cycle time and precedence restrictions:

$$L_i = [(T_i + \sum_{k \epsilon P_i} T_k) / C]^+ \qquad \text{for } i=1,\ldots,N \qquad (4)$$

The bounds provided by (3) and (4) are similar to those given in [15]. We use the bounds somewhat differently, however. In [15], the bounds are used to reduce the number of 0-1 variables required in problem formulation. We use these bounds directly to specify the range of integer values A_i can assume in assigning task i to a work station.

Stronger bounds than those given by (3) and (4) may be found when the magnitude of C approaches the maximum task time, T*. This is fortunate, since our experience indicates that a problem becomes increasingly difficult to solve when the number of stations in which assignments can be made increases. This, of course, is precisely what happens when C decreases, approaching T*.

When C approaches T*, many of the tasks with performance times in magnitude close to T* typically must be assigned to a station alone. To identify such tasks, calculate U_i and L_i using (3) and (4). Then, taking each task i in order, from 1 to N, determine if there exists a task k with $T_k > 0$ such that $T_i + T_k \leq C$, where $i \neq k$. Note that the only k's that need to be

considered are those where $L_k \leq U_i$ or $U_k \geq L_i$, and either $\{k \in P_i^*$ or $k \in S_i^*\}$, or $\{k \notin P_i$ and $k \notin S_i\}$. If k does not exist, then set $T_i = C$. When all tasks have been evaluated in this fashion, recalculate U_i and L_i using (3) and (4). For the example problem in Figure I, the sets of lower and upper bounds as calculated by (4) and (3) are {1,2,2,2,3,3,3,3,4} and {4,4,5,7,7,7,7,7,7}, when H = 8. The corresponding improved lower and upper bounds obtained by using the procedure described are {1,2,3,3,3,4,4,4,5} and {3,4,5,7,7,7,7,7,7}.

It is interesting to note that for the example problem, use of this improved procedure increases the lower bound number of work stations from four to five. Since a heuristic solution of $A_N = 5$ is easily obtained, the revised lower bound verifies that the heuristic solution is optimal. This improved bounding method can thus aid the solution procedure by providing an improved bound on the minimum number of work stations required, as well as by reducing the number of station assignments which have to be explicitly evaluated for tasks when C approaches T*.

Optimizing Procedure

Our adaptation of the Balas algorithm can be summarized as follows:
(1) nonnegative integer variables A_i are used, rather than 0-1 variables;
(2) the order in which variables are considered for augmentation is prespecified by the node numbering scheme, rather than by using binary infeasibility tests;
(3) we exploit problem structure to expedite fathoming and backtracking.

Initially, a unique terminal dummy task with $T_N = 0$ is appended to the network. Lower and upper bounds are then computed, and the idle time vector, I, is initialized to C for all stations j, j = 1,2,...,(H-1). In developing the upper bounds, U_i, H is set equal to N if a heuristic solution is not known. B_N is initialized to H regardless of how H is determined.

Augmentation then begins. Task 1 is assigned to station 1: $A_1 = 1$, and I_1 is reduced by T_1. Next, task 2 is assigned to its earliest precedent and cycle-time feasible station. The earliest precedent feasible station is $j^* = \max\{A_i | i \epsilon P_2^*\}$; or $j^* = 1$, if $P_2^* = \{\phi\}$. The earliest cycle-time feasible station is determined by scanning I from j^* to U_2 for the lowest numbered station j' such that $I_{j'} \geq T_2$. Task 2 is then assigned to station j' by setting $A_2 = j'$ and subtracting T_2 from $I_{j'}$. Tasks 3 through N are assigned in similar fashion.

If augmentation proceeds to assign task N to a work station (i.e., an improved solution is found), U_i are redefined by subtracting $(B_N - A_N)$ from U_i, for $i = 1,\ldots,N$. The incumbent best solution then becomes $B_i = A_i$, $i = 1,\ldots,N$. I_j is initialized to C, $j = 1,\ldots,A_N-1$ and augmentation again begins with task 1.

However, augmentation may not proceed to task N. As augmentation progresses, for some task i there may not exist a station j in the interval $[\max\{A_k | k \epsilon P_i^*\}, U_i]$ where $I_j \geq T_i$, because of the tightened upper bounds. Consequently, backtracking occurs: $I_{A_{i-1}}$ is increased by T_{i-1}, and stations $(A_{i-1} + 1)$ to U_{i-1} are scanned for the lowest numbered station j^* such that $T_{i-1} \leq I_{j^*}$. If j^* doesn't exist, backtracking proceeds to task i-2, and so on. If j^* exists, then $A_{i-1} = j^*$ and I_{j^*} is reduced by T_{i-1}. Optimality is then assured either when $A_N = L_N$, or when an attempt is made to backtrack below job one. In the terminology of implicit enumeration, the latter is equivalent to failing to complete a partial solution in which the left-most variable has been complemented and underlined [4].

4. Exploiting Problem Structure

The ALB problem has a well-defined structure which can be exploited to improve the performance of the basic algorithm. In addition, our formulation of

the problem imposes further structure which can lead to improvements. We now describe four such improvements which reduce the computation time required to solve a given problem.

Backtracking Using Chains

Backtracking can be expedited when chains exist in an assembly network. A chain is an ordered set of consecutively numbered tasks such that each task is an immediate predecessor of the task which follows it in the set. As an example, Figure I has two chains, $\{1,2,3\}$ and $\{8,9\}$. If the search for a feasible station assignment fails for task n, ordinarily backtracking proceeds to task n - 1: task n - 1 is reassigned to a station beyond A_{n-1}, and the search for a feasible assignment for n begins anew. But if $(n-1)\varepsilon P_n^*$, then the reassignment of n - 1 to a station later than A_{n-1} only reduces the search interval for task n. Task n, in this case, can only fail again to be assigned earlier than A_n.

When chains exist, a considerable amount of searching can be avoided by directly backtracking to the highest numbered task not in the chain. Using the chain from Figure I, it can be seen that if task 3 fails to be assigned, then backtracking can proceed immediately to the fictitious task number 0; i.e., if task 3 cannot be assigned, then the incumbent solution B_N is optimal.

Backtracking Using Network Cuts

In order to describe additional means for improving algorithm efficiency, an artifice called a network cut will be introduced. This is a notion that has proven to be useful in resource-constrained project scheduling [18, 19], and draws its name from the manner in which a problem is portrayed on a Gantt chart. The notation used in describing a cut is given in Table II.

Table II

Notation Used to Describe Network Cuts

Symbol	Description
$A^{(i)}$	The current partial schedule of task assignments. $A^{(i)} = \{A_1, A_2, \ldots, A_i\}$, when task i + 1 is being considered for station assignment.
i_s	$i_s = \max\{i \mid L_i \leq s\}$.
i_s^*	$i_s^* = \max\{i \mid i \in Q_s$ and where A_i was $\leq s$ at least once since task i - 1 was last assigned$\}$.
Q_s	The ordered set of all tasks 1 through i_s. That is, $Q_s = \{1, 2, 3, \ldots, i_s\}$.
s	Cut s: a station number in the interval $[1, L_N)$, which which meets conditions C1 and C2, as specified.
$Y_s(z)$	The zth saved partial schedule at cut s, where $z = 1, 2, \ldots, Z$. That is, $Y_s(z) = A^{(i_s)}$, for some z.

A cut s is a station number that (a) is used to identify when certain fathoming and backtracking rules may be applied; and (b) is a parameter used in the application of the rules themselves. Station s is a cut if there exists a task i such that:

C1. $L_i = s + 1$.

C2. For all $k > i$, $L_k > s$.

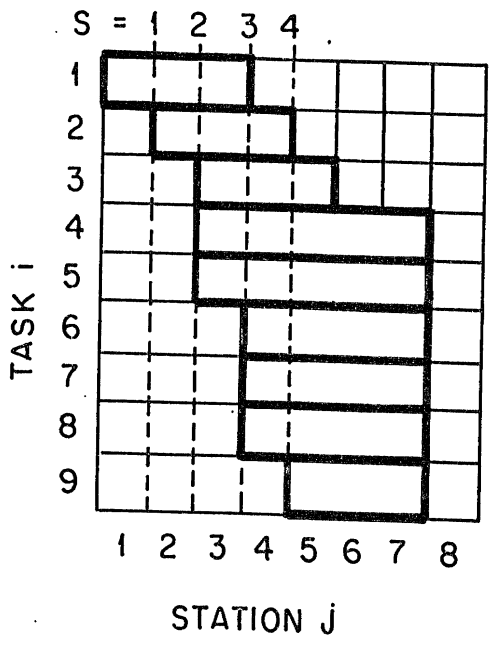
Figure II illustrates the application of C1 and C2 to the example problem. Open bars extend from L_i to U_i for each task i, representing all possible station assignments as defined by the improved bounds. Cuts are identified by the dashed lines at stations 1, 2, 3, and 4.

Figure II also illustrates the computation of i_s and Q_s. For each s, s=1,2,3,4, the set Q_s consists of all tasks 1 through i_s where i_s is the largest task number such that the earliest station into which the task can be assigned is less than or equal to s. For example, $i_1 = 1$, $Q_1 = \{1\}$; $i_2 = 2$, $Q_2 = \{1,2,\}$; $i_3 = 5$, $Q_3 = \{1,2,3,4,5\}$; $i_4 = 8$, $Q_8 = \{1,2,3,4,5,6,7,8\}$. Although not shown in Figure II, it is possible that some task k in the interval $[1, i_s]$ will not have the earliest station into which it can be assigned less than or equal to s. That is, $L_k > s$. Such tasks are included in Q_s even though their earliest station assignments are greater than s. Including all tasks 1 through i_s in Q_s is required so that optimum solutions are not excluded when employing the cut rules for fathoming partial solutions.

We shall now explore several ways in which cuts may be used to expedite backtracking and fathoming. First, we will consider backtracking. Suppose backtracking has progressed to task i_s for some s. It is then possible to immediately backtrack to task $i' = \max\{i \mid i \in Q_s \text{ and } A_i > s\}$, because backtracking

Figure II

Gantt Chart of Bowman's
Problem Illustrating Network Cuts



to any task $i'' \varepsilon Q_s$, where $i'' > i'$, could only result in reassignments for $i'', i'' + 1, \ldots, i_s$, that either have no effect on idle time I_j for $j > s$, or else decrease I_j for $j > s$, from that which was available before backtracking to i_s. (See Appendix for proofs.)

We note that this backtracking procedure is a function of the current partial schedule of task assignments $A^{(i)}$, whereas backtracking utilizing chains is dependent only on the task numbering scheme. Hence, these procedures complement one another.

Fathoming Using Cuts

Fathoming rules are considered next. Ordinarily, $A^{(i)}$ is fathomed when task $i + 1$ fails to find an $I_j \geq T_{i+1}$, where j is in the interval $[\max\{A_k | k \varepsilon P^*_{i+1}\}, U_{i+1}]$. If $i = i_s$, then two additional fathoming techniques may be applied. The first makes use of the fact that if $A^{(i)}$ is to lead to an improved solution, $A_N < B_N$, then (5) must hold.

$$\sum_{j=1}^{s} I_j \leq CU_N - \sum_{m=1}^{N} T_m \qquad (5)$$

This simply states that the idle time incurred prior to cut s cannot exceed the total idle time permitted by the current upper bound on the problem.

Rule (5) is applied immediately following the feasible assignment of task i_s. If it fails, $A^{(i_s)}$ is fathomed and backtracking proceeds to $i_s - 1$, since the reassignment of i_s to a later station will not satisfy (5). If (5) is satisfied, then augmentation continues with $i_s + 1$. In either event, the computational cost of invoking (5) is minimal, since the right-hand side of (5) is a constant, and the left is a simple sum.

A second fathoming technique relies on the generation and comparison of partial schedules in a manner analogous to their use in [19]. Basically, the current schedule $A^{(i_s)}$ is compared to a previously saved schedule at i_s, $Y_s(z)$ for some z, z=1,...,Z. If it is determined that $A^{(i_s)}$ could potentially result in an improved solution for the problem, then $A^{(i_s)}$ is saved in $Y_s(z)$, for some z, and augmentation continues with $i_s + 1$. In this case, $A^{(i_s)}$ is called a good partial schedule. However, in many cases it can be shown that $A^{(i_s)}$ cannot lead to an improved solution. When this occurs, $A^{(i_s)}$ is called an inferior partial schedule: backtracking progresses to task i_s.

Following the development in [19], a partial schedule $A^{(i_s)}$ is inferior to a saved schedule $Y_s(z)$, if (6) holds.

$$A_m = A'_m \text{ for all } m \varepsilon Q_s \ni A'_m > s$$
$$\text{where } A_m \varepsilon A^{(i_s)} \text{ and } A'_m \varepsilon Y_s(z). \qquad (6)$$

This simply indicates that if the only difference between the partial schedule $A^{(i_s)}$ and the saved schedule $Y_s(z)$ is that tasks in $Y_s(z)$ that were assigned to stations $s' \leq s$ are now assigned in $A^{(i_s)}$ to stations s' where $s' \leq s$ or $s' > s$, then an improved solution to the problem cannot result. The reason is that the rearrangement of task assignments to stations s and before cannot provide improved assignment opportunities for tasks i_s+1, i_s+2,...,N over those which were available when $Y_s(z)$ was fathomed. And, reassignments later than s merely reduce the station time that tasks i_s+1, etc., compete for.

The number of good partial schedules generated for most problems is very large. Hence, there is a need to find a mechanism for specifying Z for each cut s. The larger Z is, the more likely it is to find inferior schedules. But storage can become a limiting factor, and there is a computational expense involved in evaluating many schedules. On the basis of experience gained with

project scheduling problems, we have set Z = 10 for all s. Operationally, the ten most recently generated good schedules are saved.

Rule (6) is invoked immediately following the feasible assignment of task i_s. If $A^{(i_s)}$ is inferior, backtracking commences with i_s. If $A^{(i_s)}$ is good, it is saved in $Y_s(z)$, and augmentation continues with $i_s + 1$.

When a schedule $A^{(i_s)}$ is found where $A_i \leq s$ for all $i \epsilon Q_s$, the ALB problem may be partitioned. We call $A^{(i_s)}$ a perfect partial schedule in this instance since it is impossible to find an improved schedule for tasks $i \epsilon Q_s$, in the sense that it could lead to an improved solution for the problem. Hence, when a perfect partial schedule is found, optimality is assured when an attempt is made to backtrack to task i_s. Thus, the problem has been partitioned into two subproblems: a solved problem for tasks $i \epsilon Q_s$, and an unsolved problem for tasks $i \notin Q_s$.

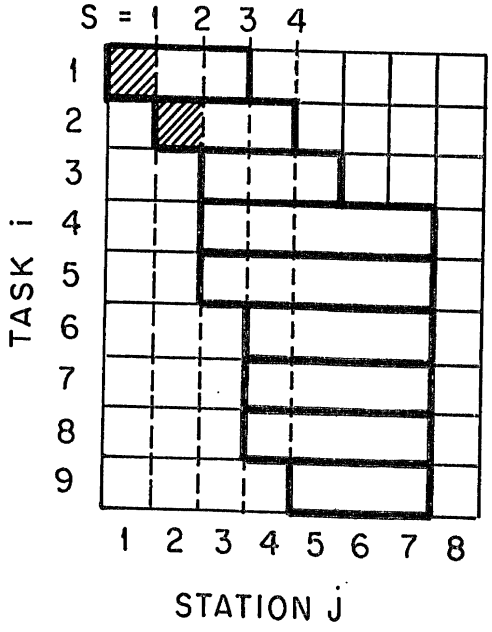
Figure III illustrates partitioning for the example problem. Here, $A^{(1_3)} = \{1\}$, and $A^{(2_3)} = \{1,2\}$. Once $A^{(2_3)}$ is specified, for example, enumeration is restricted to tasks 3,...,9. If an attempt is made to backtrack to task 2, optimality is assured. (For this trivial problem, optimality is actually verified by comparison to the improved lower bound, i.e., when $A_9 = L_9 = 5$.)

Additional Tests Using Idle Time

In this section we introduce two additional expediting rules which are based on a knowledge of the minimum amount of idle time which must be present in each work station. The first rule discussed is a "look ahead" rule which permits the algorithm to fathom a partial solution when it can be shown that the idle time incurred in the partial solution $A^{(i_s)}$ exceeds permissible levels for an overall improved solution. This rule is thus a stronger version of (5). The second rule permits improved backtracking from a cut when it can be shown that the maximum level for the partial solution idle time has been exceeded.

Figure III

Illustration of a Perfect Partial Schedule



By solving the following knapsack problem it is possible to determine the minimum amount of idle time, M_j, which must be present in each station j.

$$\text{Minimize:} \quad C - \sum_{i \in W_j} T_i X_i = M_j \tag{7}$$

$$\text{Subject to:} \quad \sum_{i \in W_j} T_i X_i \leq C \tag{8}$$

where,

$$X_i \begin{cases} = 1 \text{ If task i is assigned to station j} \\ = 0 \text{ Otherwise} \end{cases}$$

Here, W_j can be obtained by simply identifying tasks i with $L_i \leq j$ and $U_i \geq j$. For example, from Figure II, $W_3 = \{1,2,3,4,5\}$.

Equation (5) can now be replaced with (9).

$$\sum_{j=1}^{s} I_j \leq CU_N - \sum_{m=1}^{N} T_m - \sum_{j=s+1}^{U_N} M_j \tag{9}$$

The stronger test provided by (9) obviously comes at some computational expense. However, the knapsack problem itself is highly structured and usually yields solutions very quickly using a simple enumeration procedure. (Tasks in W_j are relabeled by non-increasing task time. Then X_i are augmented to 1 in order of task number. Backtracking is in reverse order of task number.) On our test problems, this idle time test has been very effective in general, but specifically when the interaction of large task times and strong precedence relationships in a problem "force" the total idle time to be distributed into certain work stations.

If the partial solution $A^{(i_s)}$ is fathomed due to excessive idle time by (5) or (9), then instead of backtracking to task $i_s - 1$, it is possible to backtrack to i_s^*. The proof of this assertion follows from the observation that it is impossible to reassign any task $i' \epsilon \{i | i > i_s^* \text{ and } i \epsilon Q_s\}$ to a station before s + 1, given the partial solution $A^{(i_s^*)}$. (If it were possible, it would

already have occurred during the enumeration process, given the convention of assigning tasks to the lowest numbered work station that is precedent and cycle-time feasible). Hence, backtracking to any task i' cannot reduce the excessive idle time occurring in stations 1 to s, which was the original reason why fathoming by (5) or (9) took place. The effect of this backtracking procedure is most pronounced when the knapsack routine identifies a relatively uniform distribution of idle time across all stations, or when total idle time is close to zero. Under these conditions, it often eliminates from explicit evaluation hundreds of partial solutions which would not have led to improved solutions to the problem.

5. Computational Experience

The algorithm described in Section 3 and each of the problem structure exploiting techniques described in Section 4 were programmed in FORTRAN IV and tested on an Amdahl 470/V8 computer (H compiler, OPT = 2). The results from applying this program to a series of test problems appearing in the open literature are given in Table III. The cycle times selected include both those that have previously appeared in the open literature, and additional times which were selected in an effort to obtain problems where the optimal solution is not equal to the theoretical lower bound. Problems with the latter characteristic were sought because without a proven lower bound a problem is generally more difficult to solve optimally. It is, of course, not always possible to select a cycle time which will give an optimal solution greater than the theoretical minimum number of stations. The Kilbridge and Wester problem, in fact, has an optimal solution equal to the theoretical minimum number of stations for the complete range of cycle times, $T^* \leq C \leq \Sigma T_i$.

Table III

Computational Results

Problem*	Number of Tasks	Cycle Time	Optimal Number of Stations	CPU Time** With Cuts	CPU Time** Basic Approach Without Cuts	CPU Time** Heur. Start With Cuts
Merten	7	6	6+	.022 (.022)	.021	.021 H
		7 L	5	.024	.023	.021 H
		8	5+	.022	.020	.022 H
		10 L	3	.022	.020	.019 H
		15	2	.016	.016	.017 H
		18 L	2	.017	.016	.015 H
Jaeschke	9	6	8+	.028 (.028)	.026	.025 H
		7 L	7+	.027	.025	.024 H
		8	6+	.023 (.026)	.020	.022 H
		10 L	4	.024	.020	.020 H
		18 L	3	.020	.020	.021 H
Jackson	11	7	8+	.030	.028	.026 H
		9	6	.025	.022	.021 H
		10 L	5	.026	.026	.026
		13	4	.027	.021	.024 H
		14 L	4	.022	.022	.021 H
		21 L	3	.025	.020	.022 H
Mitchell	21	14 L	8	.055	.048	.049 H
		15	8+	.039	.076	.038 H
		21	5	.045	.044	.042
		26 L	5	.039	.035	.039 H
		35	3	.037	.038	.039
		39 L	3	.033	.033	.032 H
Heskia	28	1.38 L	8	.059	.058	.046 H
		2.05	5	.867	6.881	.044 H
		2.16 L	5	.041	.058	.041 H
		2.56	4	.046	.050	.043 H
		3.24 L	4	.041	.040	.040 H
		3.42	3	.039	.041	.036 H
Sawyer	30	25	14+	1.106	> 8	1.100
		27 L	13+	.524	> 8	.522
		30	12+	1.555	> 8	.169 H
		36	10+	.048	> 8	.046 H
		41 L	8	.292	6.052	.293
		54	7+	.048	> 8	.050 H
		75 L	5	.043	.047	.042 H
Kilbridge & Wester	45	57 L	10	5.828	> 8	.082 H
		79	7	.221	> 8	.218
		92	6	.216	> 8	.215
		110 L	6	.063	.062	.063 H
		138	4	.089	.473	.086
		184 L	3	.071	.067	.064
Tonge	70	176 L	21+	> 8	> 8	> 8
		364 L	11	.100	.100	.102
		410	9	.101	.099	.098 H
		468 L	8	.100	.097	.097 H
		527	7	.098	.099	.094 H

Table III (continued)

Computational Results

Problem*	Number of Tasks	Cycle Time	Optimal Number of Stations	CPU Time**		
				With Cuts	Basic Approach Without Cuts	Heur. Start With Cuts
Arcus	83	50.48	16+	.470	> 8	.281 H
		58.53 L	14	.127	.122	.125 H
		68.42 L	12	.126	.125	.123 H
		75.71	11+	.126 (.527)	.122	.126 H
		84.12	10+	.750	> 8	.774 H
		88.98	9	.126	.118	.121 H
Arcus	111	108.16	8+	.254	> 8	.251 H
		57.55 L	27	> 8	> 8	.180 H
		88.47	18+	1.921	> 8	.171 H
		100.27	16+	6.101	> 8	3.142 H
		107.43 L	15+	2.883	> 8	2.880 H
		113.78 L	14	.166	.163	.160 H
		170.67 L	9	.162	.161	.159 H

* Problem sources are given in the references. See the Tonge reference for a description of the Mitchell problem.

** Amdahl 470/V8 CPU time in seconds, including input and output, to find and verify the optimal solution.

+ The optimal number of stations is greater than the theoretical minimum number of stations. For the four problems reporting a time in parentheses, the optimal solution equals the improved lower bound. The number in the parentheses indicates the solution time when the improved lower is not used.

L The cycle time previously reported in the literature.

H The heuristic solution is also optimal.

The cycle times previously appearing in the literature are indicated by an "L" in Table III. Optimal solutions greater than the theoretical minimum number of stations are noted by a "+". Four of the sixty problems have optimal solutions equal to our improved lower bound. These problems are identified as having three solution times, with the number in parentheses the time required when this improved bound is not used by the optimizing procedure. The solution times reported are total CPU times in seconds for reading the problem data, finding and verifying the optimal solution, and printing solution results.

Each problem-cycle time combination was solved with and without the cut related exploiting rules as indicated by the nomenclature "With Cuts" and "Basic Approach Without Cuts." In all cases, no initializing heuristic solution was employed: H was initially set equal to N and the algorithm worked "down" from this solution to the optimal solution. With the exception of the Mitchell (C = 15) problem, there is little to differentiate the basic approach results from the basic approach with the cut features appended for the Merten, Jaeschke, Mitchell and Heskia problems. This is consistent with our experience in general: The basic approach provides cost-effective optimal solutions to problems with up to about 25 tasks. For problems in this range, the computational overhead involved in invoking the cut-related rules usually balances the benefits derived from the reduction in enumeration the rules afford. Hence, the solution times with and without cuts are usually roughly equivalent for problems of this size. (Several runs of the program on the same set of problems yielded time variations of ± .003 seconds. Hence, care should be taken in inferring algorithmic efficiency differences when solution time differences are less than .006 seconds.)

For problems containing more than 25 tasks, the power of cut-based rules becomes more evident. For many of the larger test problems the optimal

solution could not be found and verified without the rules within the maximum CPU time permitted of 8 seconds, although they could be solved well within this time when cut rules were invoked. The extreme example of this is the Sawyer problem (C = 36 and C = 54).

Within the eight second imposed time limit optimal solutions could not be found and verified for only two problems, the Tonge-70 (C = 176) and Arcus-111 (C = 57.55). In the former case the optimal solution of 21 was actually found within .072 seconds, but it could not be verified as optimal within 8 seconds. In the latter case, the best solution found was 29 stations before the program was terminated.

These results are very encouraging and compare favorably with results obtained from the branch and bound algorithm recently developed by Magazine and Wee [11, 12, 13]. In Table 1 of [13] they report solution times (using an IBM (VM) 370), for 27 of the 60 problems noted by an "L" in our Table III. (They did not solve the Arcus-83 problem.) They obtained optimal solutions to all 27 problems, but did not verify optimality for the Sawyer (C = 27) and Tonge (C = 176) problems. They solved the Sawyer and Tonge problems heuristically in .316 and 3.013 seconds, respectively. The Arcus-111 (C = 57.55), problem, which we were not able to solve optimally in 8 seconds, was solved in .396 seconds with their branch and bound algorithm. They also solved three other problems using heuristic rules, Mitchell (C = 14), Sawyer (C = 41) and Kilbridge and Wester (C = 57), but were able to verify optimality in each case, because the heuristic solution is equal to the theoretical minimum number of stations. The total reported times for the 26 problems (i.e., without Arcus C = 57.55) for which both approaches obtained (but may not have verified) optimal solutions is 7.053 seconds from [13] and is 7.824 seconds from Table III, plus the Tonge (C = 176) time of .072.

These time comparisons are not intended to demonstrate that one approach is faster (better) than another. Such an inference would be unfounded since the times are reported for different computers and timing routines. Rather, it is to demonstrate that both are effective procedures for obtaining optimal solutions for many realistically sized ALB problems.

Other attractive features of our approach are not evident from the data presented in Table III. First, our approach requires only a modest amount of computer storage. For example, the program used in this paper requires less than 140 K bytes of storage and is dimensioned to solve problems with up to 120 tasks where each task may have up to 25 immediate predecessor tasks. (If necessary, storage requirements could further be reduced by the elimination of such user-convenience items such as Gantt charts.)

Second, our procedure does not require the user to set a critical limit or implement other rules for eliminating portions of the search as is frequently the case in employing dynamic programming or branch and bound methods to solve this problem. This is because the exact storage requirements are known in advance with our procedure, and this is typically not the case with these other approaches. Furthermore, the performance of our approach is frequently improved through the use of well-known heuristics such as rank positional weight, maximum task time first (called IUFFD in [13]), and others [17]. Initializing our procedure with a simple heuristic solution, rather than setting $H = N$, reduces the upper bounds U_i, which in turn reduces the feasible search intervals for each task. Using a heuristic start typically results in faster solution times because the intervals are smaller, or because the heuristic solution can be verified as optimal directly through a bounds test. Also, since a heuristic such as maximum task time first requires very little computational effort and little computer core to program, the costs of using such a technique are negligible.

In Table III under the column "Heur. Start With Cuts," total CPU solution times are given when the procedure is initialized with a heuristic solution provided by a rule called "upper bound first" [17]. This rule assigns precedent and cycle time feasible tasks to work stations in order of nonincreasing U_i (instead of nonincreasing T_i as does maximum task time first rule). The CPU times shown include the additional time required to obtain the heuristic solution. Using this heuristic initializing procedure, our routine with cut-based rules found and verified optimal solutions to 59 of the 60 problems in a total time of 12.660 seconds. The optimal solution to the Tonge (C = 176) problem was found in .072 seconds as before, but was not verified optimal within 8 seconds.

6. Conclusions

An integer programming algorithm for determining the minimum number of work stations to assemble a product subject to precedence and cycle time constraints is described. The algorithm is based on a systematic evaluation of all possible assignments of work tasks to a work station. Network cuts, idle time tests, and the use of network chains greatly enhance the efficacy of the approach described. Computational results on a series of test problems found in the open literature are reported and are quite favorable. The general conclusion to be reached is that the procedure can obtain optimal balances in a reasonable amount of time for assembly lines consisting of fifty or fewer tasks. While the algorithm is capable in certain instances of optimally solving assembly lines with one hundred or more tasks, such problems are generally characterized by a large cycle time in relation to the maximum task time. For those problems in which the cycle time is greater than 125 to 150 percent of the maximum task time, for example, the procedure is likely to produce an optimal solution in a reasonable amount of computation time.

Acknowledgements

The authors express their appreciation to the referees for providing a number of constructive comments on this paper. Our special thanks go to Michael Magazine, University of Waterloo, for his careful reading of the drafts and his insightful suggestions for improving the final manuscript.

Appendix

Theorem 1

If $A^{(i_s)}$ is inferior to any saved schedule in $Y_s(z)$, $z = 1,\ldots,Z$, then it is impossible to assign tasks $i_s + 1,\ldots,N$ such that $A_N < B_N$.

Proof

We first note that to be saved in $Y_s(z)$, a schedule must have been considered good earlier in the enumeration process. Following the saving of $Y_s(z)$ the enumeration procedure continued augmenting with tasks $i_s + 1$, etc., until $Y_s(z)$ was fathomed either because an improved solution to the problem was found or backtracking took place. Ultimately a new schedule $A^{(i_s)}$ for tasks $m\varepsilon Q_s$ was calculated. If (6) holds, then there are identical assignments in $A^{(i_s)}$ and $Y_s(z)$ for all tasks m that had assignments $A'_m > s$ in $Y_s(z)$. Thus the only tasks that are of interest, that is those that could potentially provide improved scheduling opportunities for jobs $i_s + 1,\ldots,N$, are tasks $m^*\varepsilon Q_s$ with assignments $A'_{m^*} \leq s$. These tasks can be reassigned to stations before ($A_{m^*} \leq s$) or after ($A_{m^*} > s$) station s.

In the former case, given the definitions of s and i_s, there does not exist a task $i > i_s$ such that $L_i \leq s$. Therefore the reassignment of any m^* to any $A_{m^*} \leq s$ cannot affect the scheduling of tasks $i_s + 1,\ldots N$. In the latter case, the assignment of any m^* to any $A_{m^*} > s$ would only decrease the station time in a station $s + 1$, $s + 2$, etc. from what it was when $Y_s(z)$ was fathomed. But this would simply make it more difficult for at least one task $i_s + 1,\ldots,N$ to be assigned. Thus, $A^{(i_s)}$ cannot lead to a solution better than that which was found with the completion of $Y_s(z)$.

Theorem 2

When a (perfect partial) schedule $A^{(i_s)}$ is found where $A_m \leq s$ for all $m \epsilon Q_s$, optimality is assured when an attempt is made to backtrack to task i_s.

Proof

Given the definitions of L_i, i_s and s, there does not exist a task $i > i_s$ that can be assigned to a station less than s + 1. If all tasks $m \epsilon Q_s$ have been assigned stations $A_m \leq s$, their reassignments could be to either stations $\leq s$ or $> s$. In the former case, the reassignment of any $m \epsilon Q_s$ cannot have any effect on the reassignment of tasks $i_s + 1$, etc., given the definitions of L_i, i_s and s. In the latter case, the reassignment of any $m \epsilon Q_s$ can only reduce the station time available in some station $> s$, thus making it more difficult to reassign at least one task $i_s + 1$, etc., within its upper bound. Since the reassignment of tasks $m \epsilon Q_s$ cannot improve the reassignment opportunities for any task $i > i_s$, but in particular task N, there is no benefit to be derived from backtracking to any task $m \epsilon Q_s$. Thus when an effort is made to backtrack to $i_s \epsilon Q_s$, the incumbent solution B_N is optimal.

Theorem 3

If backtracking proceeds to i_s, then it is possible to backtrack to task $i' = \max \{i \mid i \epsilon Q_s \text{ and } A_i > s\}$.

Proof

The reason that backtracking proceeded to i_s was because it was impossible to feasibly assign some task $i > i_s$ within its upper bound U_i. Thus to improve the assignment opportunity for such a task i, backtracking

to $i'' \leq i_s$ must increase the time available for assignment in at least one station $> s$. The proof follows by showing that this time cannot be increased by backtracking to any $i'' > i'$.

Backtracking to any task $i'' \varepsilon Q_s$, where $i'' > i'$ results in the reassignment of $i'', i'' + 1, \ldots, i_s$ to either a station $\leq s$ or a station $> s$. In the former case there is no effect on the time in stations $> s$. In the latter case, the time available is actually decreased in at least one station $> s$.

Theorem 4

If the partial solution $A^{(i_s)}$ is fathomed due to (5) or (9), then it is possible to backtrack directly to i_s^*.

Proof

If (5) or (9) causes backtracking it is due to excessive idle time in stations $\leq s$, or equivalently, it is due to too much task time being assigned to stations $> s$ in the current partial solution $A^{(i_s)}$. Thus, the only way to satisfy (5) or (9) is to reassign at least one task $i \varepsilon Q_s$ that currently has an assignment $A_i > s$ to a station $\leq s$. The Theorem follows by showing that it is impossible to reassign a task $i > i_s^*$ to a station $\leq s$ until backtracking progresses to i_s^*.

The key to understanding the Theorem and the proof is to understand how the algorithm assigns tasks and backtracks. First, tasks are considered for assignment in increasing numerical order and are assigned to the earliest precedent and cycle-time feasible station. By the definition of i_s^*, this means that when i_s^* was last assigned, tasks $i_s^* + 1$, $i_s^* + 2, \ldots, i_s$ were all, in turn, assigned to stations $> s$, because their earliest feasible station assignments were $> s$. It will be shown now that backtracking to any

$i \in R_s = \{i_s^* + 1, i_s^* + 2, \ldots, i_s\}$ cannot decrease the earliest feasible station assignments to $\leq s$ for any $k \in R_s$.

In general, backtracking proceeds in reverse numerical order and since assignment is in increasing numerical order, backtracking to any task i can only affect the reassignments of tasks $\geq i$. When the algorithm backtracks to any task i, it attempts to reassign this task from A_i, its current assignment, to A_i', where $A_i < A_i' \leq U_i$. Assuming the assignment is made, the time available in station A_i is increased by T_i units and decreased by T_i units in A_i'. Of particular interest is that the reassignment of i has no impact on the time available for task assignments in stations $< A_i$. Specifically, it does not increase the time available in stations $< A_i$ for tasks $> i$.

Assume now that the partial solution $A^{(i_s)}$ has been fathomed due to (5) or (9) and that we backtrack directly to any $i' \in R_s$. Further assume that it is possible to reassign i' to A_i' where $A_{i'} < A_{i'}' \leq U_{i'}$. (If it is not possible to reassign i', then we backtrack to $i' - 1$, $i' - 2$, etc., until we find a task that can be reassigned, or backtracking proceeds to i_s^*. In the latter case the theorem is trivial. So we assume that a task is found that can be reassigned. We will still call it i', to keep the notation simple.)

Because $A_{i'} > s$, the discussion on backtracking makes it clear that i' cannot be reassigned to a station $\leq s$, nor does its reassignment have any impact on the time available in station $\leq s$. Because there has been no increase in the time available in station $\leq s$, and since the previous earliest station assignment for $i' + 1$ was $> s$, $i' + 1$ cannot now be reassigned to a station $\leq s$. By induction neither can $i' + 2$, $i' + 3$, $\ldots, i_s$ be reassigned to stations $\leq s$. Hence, the task time assigned to stations $> s$ cannot be reduced without backtracking to i_s^*, or lower, which proves the Theorem.

REFERENCES

[1] Arcus, A. L. "An Analysis of a Computer Method of Sequencing Assembly Line Operations." Ph.D. Dissertation, University of California, 1963.

[2] Balas, E. "An Additive Algorithm for Solving Linear Programs with Zero-One Variables." Operations Research 13, no. 3 (July-August 1965): 517-546.

[3] Bowman, E. H. "Assembly Line Balancing by Linear Programming." Operations Research 8, no. 3 (May-June 1960): 385-389.

[4] Goeffrion, A. M. "An Improved Implicit Enumeration Approach for Integer Programming." Operations Research 17, no. 3 (May-June 1969): 137-151.

[5] Gutjahr, A. L. and Nemhauser, G. L. "An Algorithm for the Line Balancing Problem." Management Science 2, no. 2 (November 1964): 308-315.

[6] Held, M.; Karp, R. M.; and Sharaeshian, R. "Assembly Line Balancing--Dynamic Programming with Precedence Constraints." Operations Research 10, no. 3 (May-June 1963): 442-460.

[7] Heskia, Heskiaoff. "An Heuristic Method for Balancing Assembly Lines." Western Electric Engineer 12, no. 3 (October 1968): 9-16.

[8] Jackson, J. R. "A Computing Procedure for a Line Balancing Problem." Management Science 2, no. 3 (April 1956): 261-272.

[9] Jaeschke, G. "Eine allgemaine Methode Zur Losung Kombinatorisher Probleme." Ablauf-und planungforschung 5 (1964): 133-153.

[10] Kilbridge, M. D., and Wester, L. "A Heuristic Method of Assembly Line Balancing." Journal of Industrial Engineering 12, no. 4 (July-August 1961): 292-298.

[11] Magazine, M. J. and Wee, T. S. "An Iterative Improvement Heuristic for Bin Packing and Assembly Line Balancing Problems." Working Paper, Department of Management Sciences, University of Waterloo, 1979.

[12] _____, "Generalization of Bin Packing Heuristic to the Assembly Line Balancing Problem." Working Paper, Department of Management Sciences, University of Waterloo, 1979.

[13] _____, "An Efficient Branch and Bound Algorithm for the Assembly Line Balancing Problem." Working Paper, Department of Management Science, University of Waterloo, 1979.

[14] Merten, P. "Assembly Line Balancing by Partial Enumeration." Ablaufund Planungforschung 8 (1967): 429-433.

[15] Patterson, James H., and Albracht, Joseph J. "Assembly Line Balancing: Zero-One Programming with Fibonacci Search." Operations Research 23, no. 1 (January-February 1975): 166-172.

[16] Sawyer, J. F. H. Line Balancing. Washington, D.C.: Machinery and Allied Products Institute, 1970.

[17] Talbot, F. Brian. "An Integer Programming Algorithm for the Single Model Assembly Line Balancing Problem." Working Paper 171, Division of Research, Graduate School of Business Administration, The University of Michigan, 1978.

[18] _____, "An Integer Programming Algorithm for the Resource-Constrained Project Scheduling Problem," Ph.D. Dissertation, Pennsylvan a State University, 1976.

[19] _____, and Patterson, James H. "An Efficient Integer Programming Algorithm with Network Cuts for Solving Resource Constrained Scheduling Problems " Management Science 24, no. 11 (July 1978): 1163-1174.

[20] Thangavelu, S. R., and Shetty, C. M. "Assembly Line Balancing by Zero-One Integer Programming." AIIE Transactions 3, no. 1 (March 1971): 61-68.

[21] Tonge, F. M. A Heuristic Program of Assembly Line Balancing. New York: Prentice-Hall, 1961.

[22] White, W. W. "Comments on a Paper by Bowman." Operations Research 9, no. 2 (March-April 1961): 274-276.