EFFICIENT MODULAR IMPLEMENTATION OF BRANCH-AND-BOUND ALGORITHMS*

Roger V. Johnson
School of Business Administration, University of Michigan, Ann Arbor, MI 48109-1234

ABSTRACT

This paper demonstrates how branch-and-bound algorithms can be modularized to obtain implementation efficiencies. For the manager, this advantage can be used to obtain faster implementation of algorithm results; for the scientist, it allows efficiencies in the construction of similar algorithms with different search and addressing structures for the purpose of testing to find a preferred algorithm. The demonstration in part is achieved by showing how the computer code of a central module of logic can be transported between different algorithms that have the same search strategy. Modularizations of three common searches (the best-bound search and two variants of the last-in-first-out search) with two addressing methods are detailed and contrasted. Using four assembly line balancing algorithms as examples, modularization is demonstrated and the search and addressing methods are contrasted. The application potential of modularization is broad and includes linear programming-based integer programming. Benefits and disadvantages of modularization are discussed. Computational results demonstrate the viability of the method.

Subject Areas: Discrete Programming, Line Balancing, Mathematical Programming, and Search Theory.

INTRODUCTION

In order to modularize branch-and-branch algorithms, the algorithms in this paper consist of three components: *data blocks* that represent the state of the algorithm computations at each node as that node is formed; *data update logic* that is required to update or create these data blocks as each new node is formed (the specifics of these two components are unique for each algorithm and are grouped in Table 1); and the *central module of logic*, common, right down to the computer implementation level, to all algorithms that have the same search strategy. The third component links the data blocks and the data update logic to guarantee that candidate problem selection converges to optimality in the manner of the search type being used. It also ensures correct data addressing, bound comparison, and incumbent (best-found) solution management, often the most difficult parts of an algorithm to code and validate. In a later section, flow charts of the central modules are provided for each of the three search methods: the best-bound search and two variants of the last-in-first-out (LIFO) search (in this paper referred to as the *deep-sea-troll* search and the *laser* search).

---

*Thanks to my University of Michigan colleague F. Brian Talbot, James H. Patterson (Indiana University) and Michael Magazine (University of Waterloo) for permission to use their codes. Thanks also to the anonymous referees and to my former colleagues at UCLA, Elwood S. Buffa and Rosser T. Nelson, for their helpful comments on earlier versions of this paper.

Table 1: Required data blocks and data update logic.

Required Data Blocks	Required Data Update Logic	Data Blocks Available to Update Logic (forming node n from node m)		
		Best-Bound Search	LIFO Deep-Sea-Troll Search	LIFO Laser Search
P (problem data)	Read data	—	—	—
NN_m (data addressed by node number m)	At source node: Initialize NN_0	P	P	P
	Form new node: Define NN_m	NN_n, P	NN_n, CP_{old}, P	NN_n, CP_{old}, P NN_{pointer}
CP (data addressed as candidate problem)	At source node: Initialize CP	Not used	NN_0, P	NN_0, P
	Form new node: Update CP	Not used	NN_m, CP_{old}, P	NN_m, CP_{old}, P
	Backtrack: Recreate old CP	Not used	$NN_m, NN_n, P,$ CP_{new}	NN_m, NN_n, P CP_{new}
B_m (objective function bound)	Compute bound	NN_m, P	NN_m, CP_{old}, P	NN_m, CP_{new}, P
T_m (terminal node indicator)	Set indicator: 0: terminal node 1: otherwise	NN_m, P	NN_m, CP_{new}, P	NN_m, CP_{new}, P
V_m (objective function value)	Compute value	NN_m, P	NN_m, CP_{new}, P	NN_m, CP_{new}, P
Pointer (indicates extension or backtracking)	Reset pointer to -1 if a new node cannot be formed during Form New Node	Not used	Not used	After backtrack, the address of the previous candidate problem
I (incumbent solution and its addresses)	Update incumbent solution: Copy $NN_{(\text{optimum nodes})}$ and CP to I	—	CP and node addresses	CP and node addresses
	Print incumbent solution	Node addresses	I	I

BENEFITS AND DISADVANTAGES OF MODULARIZATION

The decision to modularize a branch-and-bound algorithm involves trade-offs. The central module includes a difficult part of the algorithm to code and verify.

However, it requires development and coding only once for all algorithms having the same search strategy. Therefore, substantial development time is saved when second and subsequent algorithms need to be coded. Since modularization requires that structured programming techniques be applied to the entire algorithm, further reductions in implementation time should be derived from increased code clarity and localization of code checking and validation.

The gains of modularization are greater for best-bound searches since these require that many more nodes be stored at any point in the computations than LIFO searches. They also require jump-tracking, causing node addressing to be more complicated. The advantages of modularization also are greater for nonbinary trees since these generally are more difficult to code than binary trees.

Offsetting these gains, some loss of computational speed occurs if the original coding of the central module constrains the choice of data structures for a subsequent algorithm. This loss would be of a linear order only since no change occurs in the portion of the tree that needs to be explored explicitly or in the number of iterations required through the use of nonoptimal data structures. If this loss exists at all, it is not likely to be serious since even binary representation of data (see [5] for how this can be done) remains possible. A second disadvantage is that to gain the full benefits of modularization, the logic of the central module should be coded for the "general" case. This requires a deeper level of conceptual understanding of algorithmic structure than is needed for any specific algorithm.

TWO ADDRESSING METHODS

In all branch-and-bound searches, a new node's data are a function of its immediate ancestor node's data and the data update logic used to move from the ancestor to the new node. To prepare for modularization, we separate the data block that defines each node into two parts according to whether the data are addressed by *node number* or by virtue of the node being the *candidate problem*. Data items formed but not used after the formation of the current node are temporary variables. They are not significant in this paper. Addressing by node number is standard in best-bound (breadth-first) searches since the candidate problem jumps from branch to branch in the partial tree of enumerated solutions. Complete data for each node in the candidate list therefore must be stored simultaneously.

In LIFO (depth-first) searches, node data need exist only when the node is the candidate problem, which itself provides a convenient address point. This is possible because the candidate problem always moves from a node to a descending node or, during backtracking, returns to a node from a descending node. Compared with node number addressing, this eliminates multiple node-storage requirements; it requires additional data update logic for backtracking to return the candidate problem to a node from its descending node.

Since addressing by node number can sometimes be employed usefully by LIFO searches, this paper considers that possibility. Further differences between the three search strategies, many of which are well known, are summarized in Table 2.

Table 2: Principle differences between the three searches.

Criterion or Characteristic	Search Method		
	Best-Bound	LIFO: Deep-Sea-Troll	LIFO: Laser
Total core requirements	Typically requires a large, unpredictable area	Moderate area needed. Area for node-addressed data usually unpredictable	Small predictable area needed
Early feasibility	Sometimes	Usually	Almost always
Data addressed by node number	Only available addressing method	Required for partial data of bounds and to define nodes	Optional, not often used
Data addressed as candidate problem	Cannot be used	Usual method, saves memory by avoiding multiple data sets	
Backtracking	Not used	Required for data addressed by candidate problem	
Number of nodes in the candidate list	Liable to be very large	Moderate but usually unpredictable	One
Core requirements per node in the candidate list	Complete node data must be kept for each node in the candidate list	All data addressed by node number must be kept for each node in the candidate list. However, only one set of data addressed by candidate problem is kept; total core requirements are minimized by maximizing use of candidate problem addressing	
Computation time per node	Slowest (complete new data set must be defined for each new node)	Slow only for data addressed by node number (often few elements). Data addressed by candidate problem computed very quickly if few items are updated for each new node; however, time for backtracking is required for candidate problem-addressed data	
Number of nodes formed	Lowest (except in unusual cases identified in [4])	More than best-bound search	More nodes (inferior node selection). In case of many tied bounds, savings can be achieved
Total computation time	No method dominates for all cases		

MODULARIZATION OF ALGORITHMS

To allow modularization, the central module's computer code must be able to operate on data from different algorithms that use the same search strategy. This is

Table 3: Data block accessibility by data update logic and central module.

Data Blocks Required	Access by Central Module	Access by Algorithm-Specific Data Update Logic
P (problem data)	Not accessed	Form defined and used
NN (data addressed by node number)	Address provided	Form defined and used
CP (data addressed by candidate problem)	Not accessed	Form defined and used
B (objective function bound)	Address provided; saved and used	Defined
T (terminal node identification)	True-or-false result saved and used	Defined
V (node objective function)	Saved and compared	Defined
Pointer (laser search only)	Defined and used	Defined and used
I (incumbent solution)	Address provided for data addressed by node number; candidate problem-addressed data not accessed	For best-bound search: defined in *NN* arrays, printed using address provided by central module For node-addressed data in LIFO searches: copied to *I* using address provided by central module and printed For candidate problem-addressed data: copied from *CP* and printed

possible when data accessibility between the central module and the algorithm-specific data update logic is as shown in Table 3.

The form and accessibility of each data block now is discussed. Neither the problem data nor data addressed as the candidate problem are addressed or accessed by the central module; their forms are unconstrained by the central module.

The central module does not operate on data addressed by node number, but it does provide the node's address. Therefore, the user can select any form of data that can use a scalar address. At the coding level, this can be any combination of scalars and arrays stored in a vector or an array of one greater dimension than is needed for data at one node. The additional dimension provides for addressing the data of multiple nodes. The central module also provides the candidate problem's predecessor's address. As an example, using FORTRAN, suppose a portion of the data consists of a scalar (SCALAR) and a two-dimensional array (TWODIM). If the value of the scalar and the (i,j)th entry of the array are to be one less than at the predecessor node, a FORTRAN implementation would be:

```
SCALAR(NEWNOD)=SCALAR(NODPRE)-1
TWODIM(NEWNOD,I,J)=TWODIM(NODPRE,I,J)-1
```

where NEWNOD and NODPRE are the addresses of the new node and its immediate predecessor supplied by the central module and I and J are addressing variables defined by the data update logic of the specific algorithm.

The objective function and bounds associated with the node use the node address. Further, the objective function and bounds are operated on by the central module and compared to the incumbent solution's objective function. These are scalar comparison operations and are valid so long as the same units (dollars, calories, etc.) are used consistently throughout the computations. As a result, the code for these comparisons remains independent of the algorithm for which the comparison is performed. Likewise, terminal node identification requires a true-false specification by data update logic of the specific algorithm; the manner in which the central module operates on the result is independent of the specific algorithm. (In fact, the objective function value and terminal node indicator are not used again after the subsequent candidate problem is selected. They could be viewed instead as temporary variables accessible by the central module.)

A "pointer" feature is required for the laser LIFO search. It is controlled, except in one circumstance, by the central module. The value of the pointer depends on the position of the search in the tree of solutions. A new node descending from a candidate problem can be formed immediately after the candidate problem itself has been formed or immediately after a backtrack to the candidate problem. In the former case, the pointer is set to 0 by the central module to signal that the node being formed is the first to descend from the candidate problem. In the latter case, the central module sets the pointer to the node address of the candidate problem immediately before the backtrack (always a positive integer). This allows the data update logic to access that node's data and to correctly form the next node descending from the candidate problem. In this case, the address of the new node will be identical to the one it replaced, a fact that sometimes can be used to advantage in obtaining faster code. If the data update logic is unsuccessful in forming a new node (which will occur when no further nodes descending from the candidate problem need to be formed), the data update logic must signal this to the central module by setting the pointer to -1.

The maintenance method of the incumbent solution depends on the search method. Since backtracking does not occur in the best-bound search, data are kept intact and it is sufficient to save only the addresses. This is done by the central module. Code must be written to print the incumbent solution, given these addresses, when optimality is reached. In LIFO searches, data of the incumbent solution usually are destroyed. Therefore, data must first be copied into the incumbent solution data area and saved. Code must be written both for this copying and for printing the final incumbent solution.

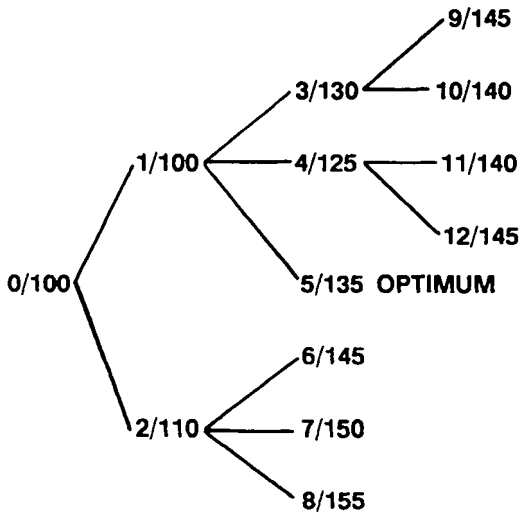
THREE CLASSES OF BRANCH-AND-BOUND SEARCH

Since gains from modularization are derived from similarity in search strategies, common logic was sought and found in three frequently used search strategies. There was little difficulty in finding the common logic in *best-bound* searches, so named

because the node possessing the best bound always is selected as the new candidate problem. But two variants of *LIFO* searches were found that differed in how they timed the formation of new nodes. These differences are pronounced when significant time or memory is required to define all the nodes descending from a candidate problem; in binary searches, in which nodes are defined easily, these differences are not important. Since these two search procedures are not yet identified as two distinct procedures in the literature, in this paper they are referred to as the *deep-sea-troll* search and the *laser* search. The choice of these names is related to those portions of the tree of enumerated solutions that each procedure stores in the computer at the time a terminal node is discovered. In the deep-sea-troll search, a string of arcs through the tree, forming a path from the source node to the terminal node, plus all unfathomed single arcs leading from that path are stored. The string of arcs represents a deep-sea-troll line; the single arcs leading from that string represent fish hooks attached to the line in clusters. These hooks lead to the portion of the tree not yet created. In the laser search, only one path through the tree is stored at any one time. Single arcs leading from the path are not stored. This direct path can be viewed as a laser beam, since the search gets to a specific terminal node via a single path, mimicking the laser beam.

In order to highlight the principal differences between these three search procedures, the sequences in which nodes are formed are shown below for the example in Figure 1. In this contrived example, all nodes in the tree are formed during each search. In Figure 1, the nodes are numbered arbitrarily, and the computed lower bound of the objective function is noted for each node.

Figure 1: Sample problem in tree form.



Note: *x/y* indicates that node *x* has an objective function bound or value of *y*. Nodes 0 through 4 represent incomplete or infeasible solutions. Nodes 5 through 12 represent feasible solutions.

The sequences of node creation for each search are as follows (parentheses indicate simultaneous node creation):

Best-bound: (1,2), (3,4,5), (6,7,8), (11,12), (9,10)
Deep-sea-troll: (1,2), (3,4,5), (11,12), (9,10), (6,7,8)
Laser: 1, 3, 9, 10, 4, 11, 12, 5, 2, 6, 7, 8

The Best-Bound Search

In a best-bound search, all nodes descending from the candidate problem are created at the same time and join the candidate list simultaneously. The node in this list with the lowest bound becomes the next candidate problem, at which time the previous (old) candidate problem is removed from the candidate list. If a candidate problem at a terminal node is found to have an objective function value less than or equal to the best bound of any node in the candidate list, it is optimal and computation ceases. The logic common to best-bound searches is contained in the flowchart of the best-bound-search central module in Figure 2.

Examples of best-bound searches include the "principle" version of Little, Murty, Sweeney, and Karel's [11] traveling salesman algorithm, the (original) Land-Doig [10] mixed integer-programming algorithm, and the "branch-and-bound" version of Greenberg and Hegerich's [6] knapsack algorithm.

The Deep-Sea-Troll Search

The deep-sea-troll LIFO search is distinguished by how it times the formation of arcs descending from any active node. When one node descending from the candidate problem is formed, all nodes descending from it are formed before any other node in the tree is formed. Further, after these nodes are formed, the next set of nodes formed will descend from one of these newly created nodes. When a terminal node is reached, its objective function value is compared with that of the incumbent solution. The new solution becomes the incumbent if it is superior. The search then backtracks along the path to the nearest node that has a descending node which has not been extended. The partial solution that has been backtracked over is discarded. If this descending node's bound is superior to the objective function value of the incumbent solution, it becomes the candidate problem and nodes descending from it are formed and bounds calculated. If the bound is inferior to the incumbent solution's objective function value, the node is discarded and the closest unexplored node becomes the candidate problem. The logic of the deep-sea-troll-search central module computations is provided in the flowchart in Figure 3.

In most deep-sea-troll searches, at least some of the data associated with each node must be addressed by node number. These data provide "hooks" to that portion of the tree yet to be explored. The remaining data can be addressed by node number or as candidate problems. The allocation of data between these two addressing methods should depend on the resulting efficiencies of required storage area and computation time. Further, since bounds of the new nodes are computed prior to selection of the new candidate problem, these bounds must be based on the data

addressed by the old candidate problem and the data addressed by the new node number. Sufficient data therefore should be placed in node-addressed data to compute the strongest possible bound for the node.

Examples of deep-sea-troll LIFO searches include the "throw away the tree" variation of Little et al.'s traveling salesman algorithm [11, pp. 983-984] and Johnson's [7] [8] assembly line balancing algorithms. Greenberg and Hegerich's [6] branch-search version of their algorithm is a LIFO search, but whether it is a deep-sea-troll or a laser search depends on the particular implementation.

Figure 2: Flowchart of the central module for the minimizing best-bound search.

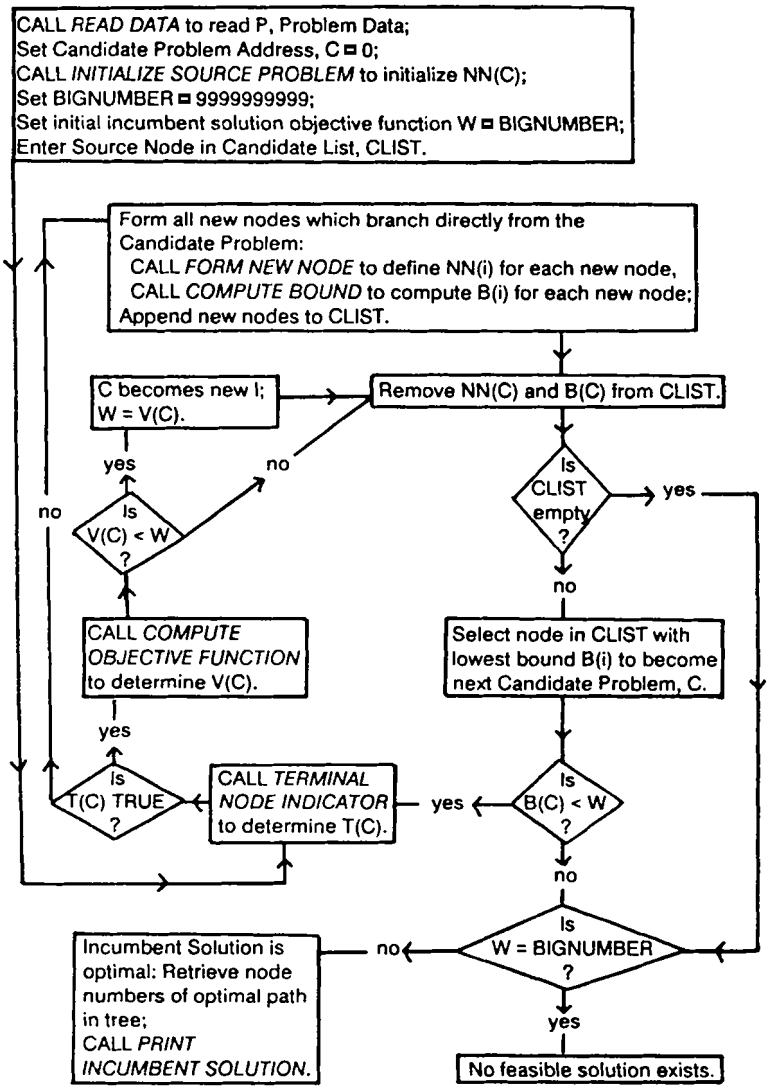
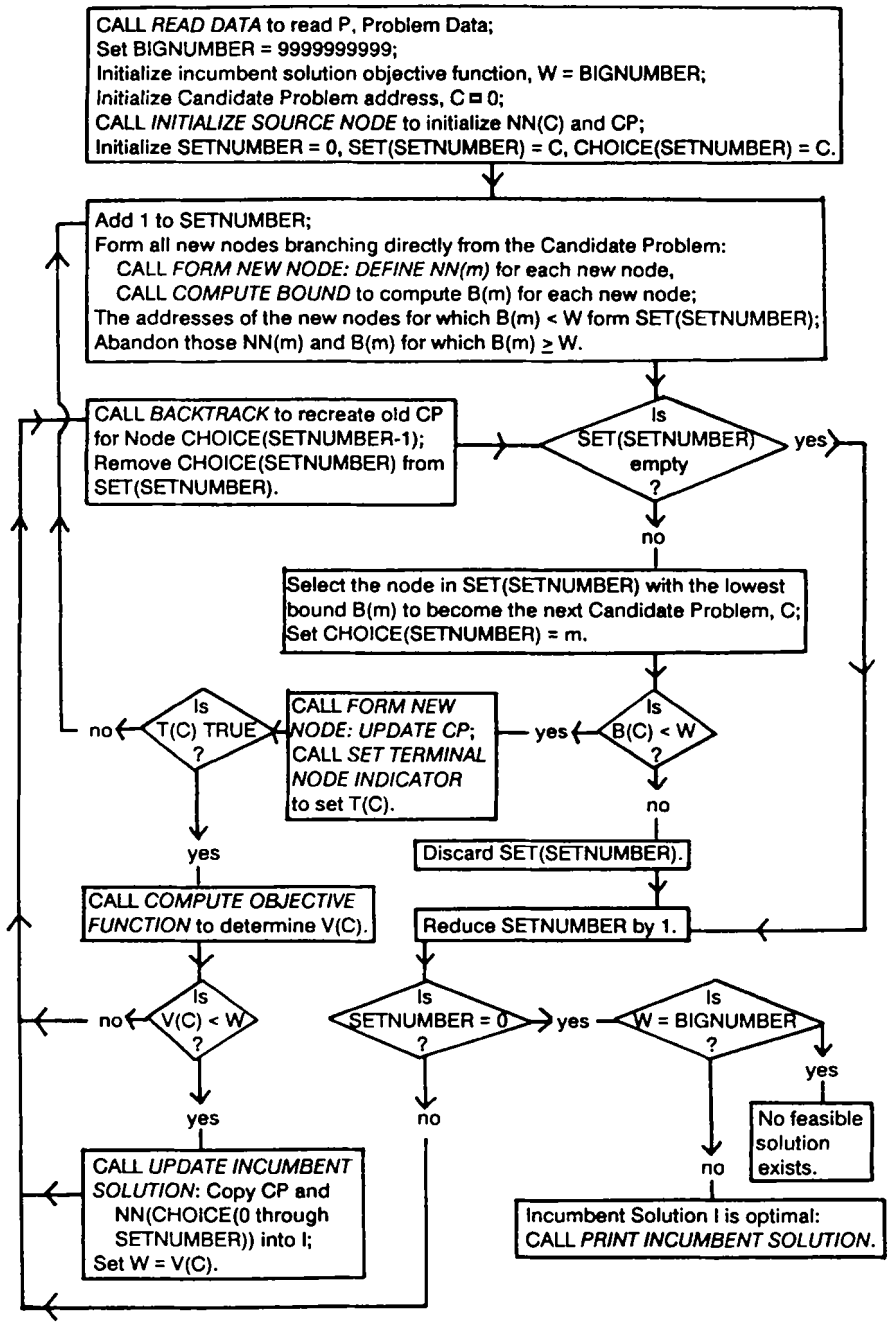


Figure 3: Flowchart of the central module for the minimizing deep-sea-troll LIFO search.



The Laser LIFO Search

The laser search is similar to the deep-sea-troll search. However, in a laser search there never is more than one node in the candidate list. Thus if two or more nodes descend from a candidate problem, only one is formed initially. The remaining node(s) is not formed until the subtree descending from the formed node, which becomes the next candidate problem, has been fathomed completely. The selection and creation of the new candidate problem therefore must be based solely on the data of the previous candidate problem (addressed either way). Thus the new candidate problem must be formed without the benefit (derived in the deep-sea-troll search) of first computing the bounds of all nodes that descend from the old candidate problem. Therefore, node selection is apt to be weaker under a laser search. Further, in order to provide information about which arcs have been formed, a pointer (as described earlier) is provided. In some searches, the pointer is disguised. For example, in a 0-1 search of the Balas [1] type, the 0-1 variable itself provides sufficient information to serve as the pointer if the first selection of 0 or 1 at each node is made consistently. A flowchart of the central module for the laser search is contained in Figure 4.

Algorithms employing a laser LIFO search include Balas's [1] 0-1 integer programming algorithm, Dakin's [3] modification of the Land-Doig [10] algorithm, Talbot's [13] project-scheduling algorithm, Talbot and Patterson's [15] assembly line balancing algorithm, and Johnson's [9] assembly line balancing algorithm.

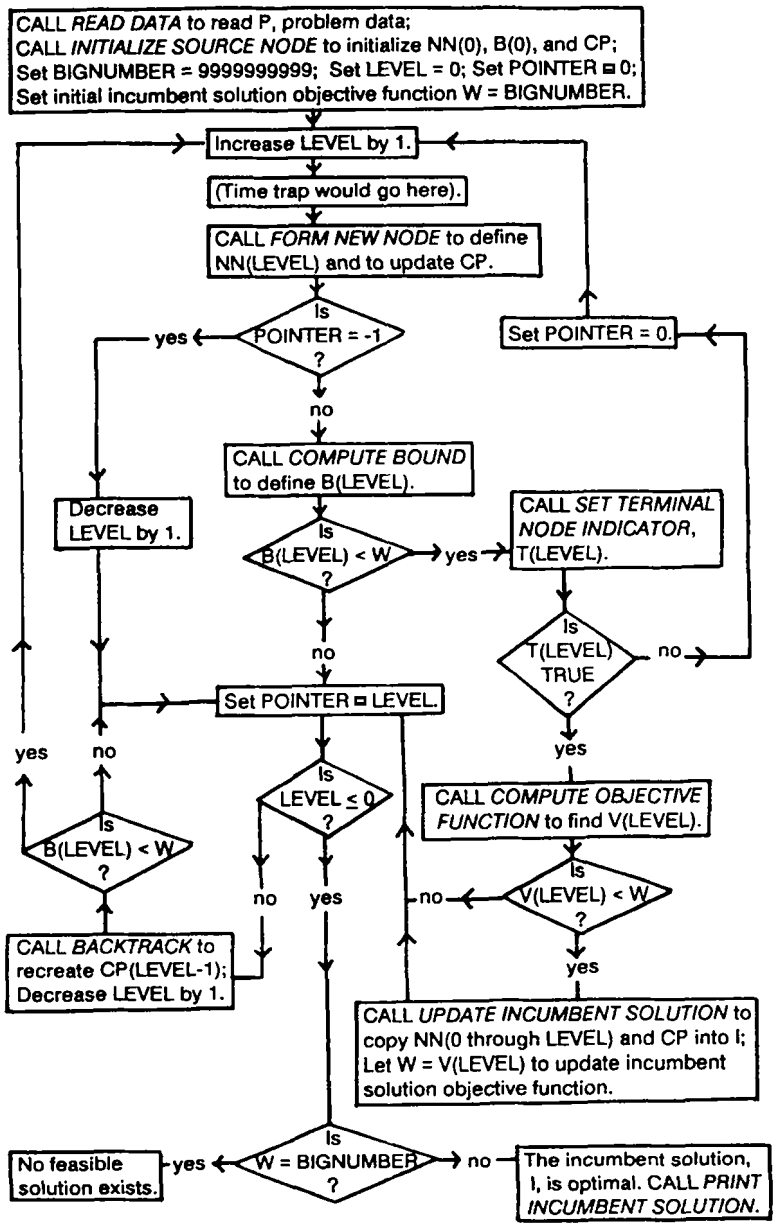
AN EXAMPLE: ASSEMBLY LINE BALANCING

In this section, four algorithms are outlined that "balance" assembly lines of the simple formulation originally described by Salverson [12]. One algorithm employs a best-bound search, two employ a deep-sea-troll search but use different addressing methods, and the fourth employs a laser search. The problem solved is: A set of tasks, each with a given performance time, is to be allocated to work stations so that the number of work stations is minimized, subject to the constraints that the time to perform the tasks allocated to each station does not exceed a given cycle time and that task precedent specifications are preserved.

In all four algorithms, each arc in the enumeration tree represents the allocation of a task to a station. Stations are filled in their physical sequence on the assembly line, so the first allocated task necessarily has no required precedents. When a subsequent task is allocated, all of its precedent tasks must already be allocated. A lower bound on the required number of stations is computed assuming that an allocation of tasks to incomplete stations exists in which these stations are filled to capacity. The bound computation of the remaining stations is the smallest integer not less than the total of task times less the free time at the current station, divided by the cycle time.

The variables that permit the four searches are defined in Table 4. The sequence and details of computations are described in the following subsections and in Table 5.

Figure 4: Flowchart of the central module for the minimizing laser LIFO search.



Best-Bound Search

This best-bound search is a simplification of the algorithm proposed by Charlton and Death [2]. Computation begins with no task allocated. Then one arc is formed descending from the source for each task with no required predecessor tasks. The node with the least bound is selected, from which the next set of new arcs is formed. This process is repeated: the node with the lowest objective function bound always is selected for the generation of the next set of descending arcs. Ties are broken naively in favor of the lowest-numbered tasks.

Table 4: Variable definitions, source node initializations, and addressing methods of assembly line algorithms.

Variable	Definition	Source Node Initialization	Addressing			
			BB	T1	T2	L
	P (Problem Data)					
$P1$	number of tasks	—	—	—	—	—
$P2$	matrix of precedent requirements: $P2_{j,i}=1$ if task j must precede task i; $=0$ otherwise	—	—	—	—	—
$P3$	cycle time	—	—	—	—	—
$P4$	$P4_i$ = performance time of task i	—	—	—	—	—
$P5$	sum of all task times	—	—	—	—	—
	Variables in node data block					
$D1_n$	task allocated at node n	0	NN	NN	NN	NN
$D2_n$	station number being built	1	NN	NN	NN	NN
$D3_{n,i}$	station at which task i is allocated; 0 means not allocated	0, all i	NN	NN	CP	CP
$D4_n$	time used at station n	0	NN	NN	NN	NN
$D5_n$	sum of allocated task times	0	NN	NN	CP	CP
$D6_i$	number of tasks not allocated that must precede task i	computed	—	—	—	CP
$D7_i$	lowest numbered task $\geq i$ that is not allocated and is available	computed	—	—	—	CP
	Objective Function Bound					
B_n	lower bound of the required number of work stations	$[P5/P3]^+$	NN	NN	NN	NN
	Terminal node indicator					
T_n	true if all tasks assigned	false	NN	NN	NN	NN
	Objective function					
V_n	value of objective function (computed only if T_n is true)	not computed	NN	NN	NN	NN

Notes: BB = best-bound search implementation; T1 = first deep-sea-troll search implementation; T2 = second deep-sea-troll search implementation; L = laser search implementation; NN = data addressed by node number; CP = data addressed by candidate problem. The subscript n is not used when data are addressed as candidate problem data.

Table 5: Assembly line algorithm computations for bounds, terminal node identification, and objective function.

Variable	Computational Formula
B_n	*Lower bound of objective function:* Best-bound search: $D2_n + [(P5 - D5_n - (P3 - D4_n))/P3]^+$ First deep-sea-troll search: $D2_n + [(P5 - D5_n - (P3 - D4_n))/P3]^+$ Second deep-sea-troll search: $D2_n + [(P5 - (D5_{old} + P4(DI_n)) - (P3 - D4_n))/P3]^+$ Laser search: $D2_n + [(P5 - D5_{new} - (P3 - D4_n))/P3]^+$ Definition: $[x]^+$ = smallest integer $\geq x$.
T_n	*Terminal Node Identification:* true only if sum of allocated task times equals total of task times: Best-bound and first deep-sea-troll search: when $D5_n = P5$ Second deep-sea-troll search: when $D5_{old} + P4(DI_n) = P5$ Laser search: when $D5_{new} = P5$
V_n	*Objective function:* $D2_n$ (computed only if T_n is true)

Each task that can fit into the partially formed station of the predecessor node p will form a separate arc. To fit at a station, task i must not have been already allocated ($D3_{p,i} = 0$), must not cause the work content at the station to exceed the cycle time ($P4_i + D4_p \leq P3$), and must satisfy the task-precedent constraints ($D3_{p,j} = 1$ for all j for which $P2_{j,i} = 1$). Further, to ensure that only combinations and not permutations of tasks are created at a particular station, the task number of a task allocated to a particular station must be greater than that of the previous task allocated to the same station ($i > DI_p$). For each of the m tasks i that satisfy these conditions at node p, an arc descending from node p is formed by defining $DI_n = i$ where $n = 1, 2, \ldots, m$.

The data update logic for forming the remainder of the data block that forms a new node n from its predecessor node p, where DI_n is allocated to the station containing DI_p, is

$D2_n = D2_p$: the station number is not changed
$D3_{n,i} = D2_n$ for $i = DI_n$: task i is allocated to station $D2_n$
$D3_{n,i} = D3_{p,i}$ for $i \neq DI_n$: other task allocations are not changed
$D4_n = D4_p + P4(DI_n)$: the station time used is updated
$D5_n = D5_p + P4(DI_n)$: the sum of allocated task times is increased

If no task fits at the current station, a new station must be started. A new node, say node n, will be formed for each task i: $DI_n = i$ for which i is not already allocated ($D3_{p,i} = 0$) and for which all task i's predecessors are allocated ($D3_{p,j} = 1$

for all j for which $P2_{j,i}=1$). The computation of variables $D3_{n,i}$ and $D5_n$ is as above, but $D2_n$ and $D4_n$ are computed as follows:

$D2_n = D2_p + 1$: a new station is formed
$D4_n = P4(D1_n)$: station time used is the performance time of the task allocated at this station

The best-bound search to balance assembly lines is described completely by the above data update logic, the variable definitions and source node initializations of Table 4; the central module of the best-bound search of Figure 2; and the logic of the bound computation, terminal node recognition, and objective function computation defined in Table 5. The linkages among these components are invoked by the CALL items shown in the flowchart of the central module in Figure 2.

For a problem with n tasks, $n+7$ memory locations are required for each node in the candidate list including bound, objective function, and terminal node recognition variables. For the data blocks selected, it is not necessary to retain information about nodes that already have served as the candidate problem. Therefore, an upper bound of the number of nodes in the candidate list is $n!$ and a maximum of $(n+7)(n!)$ memory locations is required.

First Deep-Sea-Troll LIFO Search

The first deep-sea-troll search utilizes addressing by node number only. It uses the same data blocks and data update logic as the best-bound search described in the previous section. Most differences between these two algorithms occur in their respective central modules which dictate the sequence in which nodes are created. Therefore, the central module shown in Figure 3 is used instead of that in Figure 2. Additionally, a copy of the current incumbent solution is kept.

In this LIFO search, the maximum number of nodes in the candidate list is $n+(n-1)+\ldots+1=n(n+1)/2$ where there are n tasks in the problem. Since $n+7$ memory locations are required for each node in the candidate list, a maximum of $(n+7)(n)(n+1)/2$ memory locations is needed.

Second Deep-Sea-Troll Search

The second deep-sea-troll search differs from the first in just one respect: the variables that comprise *D3* and *D5* are addressed to the candidate problem rather than by node number. As a result, data update logic to move the candidate problem from a node to its descendant node are the same as for the first deep-sea-troll search except the node subscripts on *D3* and *D5* are not used and, of the n variables in *D3* for an n-task problem, only the task being allocated needs to be updated. The timing of the computations differs according to addressing method.

Also, backtracking data-update logic must be defined for data addressed to the candidate problem, which is:

$D3(Dl_n)=0$: task Dl_n is no longer allocated
$D5 = D5_{old} - P4(Dl_n)$: the sum of allocated task times is decreased

The bound computations must be modified in this search since the variables addressed as the candidate problem (*D3* and *D5*) are not updated when the bound is computed. This is reflected in the formula in Table 5.

When a terminal node is identified as the best found, both the node addressed and the candidate problem-addressed data are saved in the incumbent solution data block.

These changes considerably reduce the computation time at each node since, as the candidate problem moves to a descendant node or backtracks, only two of the $(n+1)$ variables contained in *D3* and *D5* (rather than all $(n+1)$) are modified. The total number of variables computed per node created, including backtracking and the six node-addressed variables, is reduced from $n+7$ (using only node addressing) to $10 (=6+2\times2)$ using this search mode's addressing. Further, some nodes in the candidate list never become the candidate problem, in which case candidate problem-addressed variables are not computed at all.

Since the maximum number of active arcs is $n(n+1)/2$, the maximum number of memory locations required can be computed as $3n^2+4n+1$, derived from $6(n(n+1)/2)$ locations for node-addressed data and $(n+1)$ locations for candidate problem-addressed data.

Laser LIFO Search

In moving the candidate problem to one of its descending nodes, the logic of the laser LIFO search will depend on whether the arc is the first to descend from that node (pointer = 0) or not (pointer > 0). If the pointer, which is set by the central module, is 0, the arc will be the same as the first arc formed by either deep-sea-troll search (i.e., the minimum-numbered available task, i); the bound computation can use the newly created candidate problem data, as indicated in Table 5.

If (pointer > 0), the arc is a second or subsequent arc formed from the candidate problem at node n and the pointer gives the address of the arc previously formed from the same node. The computations depend on whether a new work station was formed for the previous arc descending from the candidate problem, recognized by $(D2_{pointer} > D2_p)$, or at an existing work station $(D2_{pointer} = D2_p)$. In either case, the lowest-numbered task $i > Dl_{pointer}$ that satisfies the conditions appropriate in the second deep-sea-troll search is selected as the new arc here: $Dl_m = i$. If no such task exists, the pointer is set to -1 to indicate that further backtracking is appropriate. The same backtracking computations are performed as in the second deep-sea-troll search.

EXPERIENCE

The advantage of the modular approach can be appreciated by considering that, with a coded central module for the best-bound search and the deep-sea-troll

search, the first two assembly line balancing algorithms described in this paper were implemented using only 38 additional FORTRAN statements. (The same statements were used for each algorithm.) A knapsack algorithm was coded using 40 additional statements. A bicriteria network cost-minimization algorithm was coded with 25 additional FORTRAN statements. (These numbers exclude statements required to input data and print the results.)

An important issue in using the modular approach is whether the modular code will be competitive with code that is fully tailored to a particular algorithm. To explore this question, the performances of seven assembly line balancing algorithms were compared. Wee and Magazine's [16] and Talbot and Patterson's [14] codes were selected to represent two algorithms that were not coded using modular methods described in this paper. (These codes appeared to be the best algorithms examined in Talbot, Patterson and Gehrlein's [15] comparative investigation of assembly line heuristics.) The two algorithms were compared with five algorithms that were coded using modular principles. Four were defined in the previous section; the fifth is Johnson's [9] algorithm. The first three described in the previous section were designed for simplicity, not computational speed. The literature set selected by Talbot et al. [15] was used. The results are shown in Table 6.

From Table 6, it can be seen that only Johnson's [9] laser search solved all problems to proven optimality, showing that any disadvantage that the modular implementation might have was not serious. More likely, the modular implementation helped. Recalling that Wee and Magazine's, Talbot and Patterson's, and Johnson's algorithms were designed primarily for speed (the latter two algorithms, being laser LIFO searches, also require small and predictable core space), we find no predictable evidence that using modular coding slows computation times. Differences in the algorithms themselves are a much more likely explanation for any computational time differences.

In comparing the four algorithms developed for this research paper, we find the algorithm employing the laser LIFO search clearly is best. It would be dangerous to extrapolate the success reported here to other problem classes. The assembly line balancing problem is characterized by many alternate optimal solutions, lessening the importance of the arc selection phase—something the laser search inherently is comparatively weak in performing. In other problem classes, node selection may be comparatively more important, indicating that if a deep-sea-troll search could extract and better utilize tighter bounds, it might lead to a better algorithm.

EXTENSIONS OF THE ASSUMPTIONS

Modularization of Other Branch-and-Bound Algorithms

Most branch-and-bound algorithms come close to the structure of the three search types detailed in this paper. However, some legitimate variations occur. For example, in the laser LIFO search, node-addressed data always are updated before candidate problem-addressed data of the same node. Since this might not always be convenient, slight changes to the central module to allow deviations might be beneficial in some instances.

Table 6: Computation comparisons of assembly line algorithms: Computation times (CPU seconds) to solve literature problems to proven optimality (on an IBM 3090 using the IBM H compiler).

Problem Name [16]	Optimal Number of Stations	Cycle Time	Talbot & Patterson [15]	Wee & Magazine [17]	Best-Bound Search	First Deep-Sea-Troll Search	Second Deep-Sea-Troll Search	Laser Search	Johnson Laser Search [9]
Merten	6	6	.00	.00	.09	.05	.03	.00	.01
	5	7	.00	.00	.10	.05	.03	.00	.00
	5	8	.00	.01	.09	.05	.03	.01	.00
	3	10	.00	.00	.09	.05	.03	.00	.00
	2	15	.00	.00	.09	.05	.03	.00	.01
	2	18	.00	.01	.09	.05	.03	.00	.01
Bowman	5	20	.01	.00	.10	.05	.03	.01	.00
Jaeschke	8	6	.00	.00	.10	.05	.03	.00	.00
	7	7	.00	.00	.09	.05	.03	.00	.01
	6	8	.00	.00	.10	.05	.03	.00	.00
	4	10	.00	.00	.09	.05	.03	.01	.00
	3	18	.00	.00	.10	.05	.03	.00	.00
Jackson	8	7	.00	.00	.17	.06	.03	.00	.00
	6	9	.00	.00	.22	.06	.03	.00	.00
	5	10	.00	.00	.13	.05	.03	.00	.00
	4	13	.00	.00	.15	.05	.03	.00	.00
	4	14	.00	.03	.16	.05	.03	.00	.00
	3	21	.00	.00	.13	.05	.03	.01	.00
Dar-El	4	48	.00	.00	.12	.05	.03	.00	.01
	3	62	.00	.01	.12	.06	.03	.00	.01
	2	94	.00	.00	.12	.05	.03	.00	.00
Mitchell	8	14	.00	.00	3. (d,f)	.08	.04	.01	.01
	8	15	.00	.01	3. (d,f)	.07	.03	.00	.01
	5	21	.00	.00	3. (d,f)	.06	.03	.01	.00
	5	26	.00	.00	3. (d,f)	.05	.03	.00	.01
	3	35	.00	.01	3. (d,f)	.06	.03	.01	.00
	3	39	.00	.00	3. (d,f)	.05	.03	.01	.00
Heskia	8	138	.00	.00	3. (d,f)	3. (c,f)	3. (c,f)	.01	.00
	5	205	.00	.00	3. (d,f)	.28	.14	.00	.00
	5	216	.00	.00	3. (d,f)	.07	.03	.00	.01
	4	256	.00	.01	3. (d,f)	.14	.07	.01	.00
	4	324	.00	.00	3. (d,f)	.05	.03	.00	.01
	3	342	.34	.00	3. (d,f)	.06	.05	.02	.01

Sawyer	14	25	3. (b,f)	.07	3. (d,f)	.87	.42	.22	.03
	13	27	3. (c,f)	.03	3. (d,f)	.75	.39	.08	.01
	12	30	3. (b,f)	1.04(a,d)	3. (d,f)	3. (b,f)	3. (b,f)	3. (b,f)	.24
	10	36	3. (b,f)	.13	3. (d,f)	3. (b,f)	1.66	.71	.08
	8	41	.00	.01	3. (d,f)	.07	.03	.02	.01
	7	54	3. (b,f)	.12(a,b)	3. (d,f)	3. (b,f)	1.67	.70	.10
	5	75	.00	.00	3. (d,f)	.06	.04	.00	.01
Kilbridge & Wester	10	57	.00	.00	3. (d,f)	3. (c,f)	3. (c,f)	.01	.01
	7	79	3. (c,f)	.03(a)	3. (d,f)	.19	.09	.18	.13
	6	92	3. (c,f)	.08(a)	3. (d,f)	.11	.06	.02	.00
	6	110	.00	.00	3. (d,f)	.07	.06	.02	.02
	4	138	.02	.11(a)	3. (d,f)	.09	.03	.04	.05
	3	184	.00	.00(a)	3. (d,f)	.06	.03	.02	.01
Tonge(e)	21	176	3. (bf)	.65(a,b)	3. (d,f)	3. (c,f)	3. (c,f)	1.73	1.58
	10	364	.01	.00	3. (d,f)	.26	.13	.01	.03
	9	410	.01	.00	3. (d,f)	.18	.11	.03	.03
	8	468	.00	.00	3. (d,f)	.14	.11	.02	.02
	7	527	.00	.00	3. (d,f)	.15	.11	.00	.00
Arcus-83	16	5048	.08	.02	3. (d,f)	3. (c,f)	3. (c,f)	.04	.04
	14	5853	.01	.00	3. (d,f)	3. (c,f)	1.72	.03	.04
	12	6842	.02	.00	3. (d,f)	3. (c,f)	3. (c,f)	.02	.04
	11	7571	.01	.00	3. (d,f)	.15	.08	.00	.04
	10	8412	.14	.03	3. (d,f)	.16	.10	.03	.02
	9	8898	.00	.00	3. (d,f)	3. (c,f)	2.22	.03	.03
	8	10816	.03	.00	3. (d,f)	.18	.08	.03	.02
Arcus-111	27	5755	.05	.00	3. (d,f)	3. (c,f)	3. (c,f)	.07	.22
	18	8847	.02	.00	3. (d,f)	3. (c,f)	3. (c,f)	.03	.06
	16	10027	2.71	.00	3. (d,f)	3. (c,f)	1.50	.03	.06
	15	10743	2.42	.10	3. (d,f)	3. (c,f)	3. (c,f)	.04	.06
	14	11378	.00	.00	3. (d,f)	1.02	.55	.03	.00
	9	17067	.00	.00	3. (d,f)	.24	.16	.04	.05
Total time (CPU secs.)			29.88	2.51	131.45	48.80	39.49	7.35	3.16
Number of problems solved in various categories:									
Proven optimality reached by algorithm			56	57	21	50	55	63	64
Proven optimality reached by heuristic			—	4	—	—	—	—	—
Optimality found but not proven			5	2	—	3	1	1	—
Optimality not reached			3	—	—	11	8	—	—
Feasibility not reached			—	1	43	—	—	—	—

Notes: (a) Dimensioned to use 1,400,000 bytes, the code converted to a heuristic to keep within available core; (b) Optimal solution found, but optimality not proven; (c) Optimal solution not found; (d) Feasible solution not found; (e) One task exceeded the cycle time of 176 and was reduced to 143 in all the computations; (f) Allocated execution time (3 CPU seconds) expired

Linear Programming-Based Methods

It is possible to use the proposed modular methods to implement integer programming algorithms based on linear programming methods. A summary is now given as an example of how this could be done for Dakin's modification of the Land-Doig procedure. The implementation is straightforward if a linear programming algorithm code that can accept (and remove) additional constraints and a coded LIFO central module are available.

A binary tree will be formed for this application. The source node will be initialized to contain the optimal solution to the linear (noninteger) problem. Node selection includes the process of selecting a noninteger variable in the optimized tableau. The two new nodes branching from the candidate problem will represent subproblems on each side of the noninteger portion of the feasible region. For example, for an integer variable $x=4.4$ in the optimized tableau of the candidate problem, the two new nodes will represent the candidate problem plus one of the additional constraints, $x \leq 4$ and $x \geq 5$. Taking the $x \leq 4$ node further, we see node-addressed data will consist of variable x, 4, and $\leq$. If this node becomes the new candidate problem, then the tableau (which is candidate problem-addressed to avoid tableau duplication) is reoptimized to include the new constraint. The bound is simply the objective function of the optimized tableau. Backtracking consists of reoptimizing after the node-addressed data (the constraint) have been removed from the tableau. The terminal node identification check will consist of verifying that all variables required to be integer do have integer values in the reoptimized tableau. The data kept in the incumbent solution will consist of the optimal solution of the problem at the appropriate node and, additionally, can include a copy of the tableau if required. The deep-sea-troll search central module should be used.

Other Search Approaches

The concept of modularization can be extended to provide for a variety of search methods including heuristic searches, hybrid branch-and-bound and dynamic programming searches, searches which learn as the search progresses, and searches which evaluate each nth-level node in a more rigorous manner than at other nodes. A central module with solution method options also may be provided. For example, the first assembly line deep-sea-troll search (which does not use the candidate problem-addressing concept) used the same code for data blocks and data update logic as the best-bound search.

The central module used offers a choice between these two searches by initializing a single flag. This entailed a more complex addressing method within the central module than either single-search central module required.

ALGORITHM IMPLEMENTATION STEPS

A logical set of steps to develop an algorithm is:

1. Generate the list of variables to define each node and the data update functions necessary to compute the variables. The size of the tree can be

kept to a minimum by appropriate definition of nodes and/or by local or global properties (such as theorems or dominance arguments) that eliminate the need to consider parts of the tree. These factors (the strength of the bound arguments, the structure of the tree of solutions that need to be enumerated, and the arc selection process) which are external to the modularization approach advocated in this paper determine the power of an algorithm.

2. Examine the variables and data update logic. Compare the computational burden and memory requirements necessary for each node. Select or develop a central module logic to match the search strategy that will achieve the algorithm power defined in 1 (above). Try to gain node-by-node efficiencies by developing an optimal or efficient data structure (with optimal data split between the two addressing methods, in the case of LIFO searches).
3. Code the central module if a code is not already available. Code modularity allows the code to be validated using the variables and data update logic of completely different algorithms. For example, the central module used for an implementation of Land-Doig mixed integer-programming code can be validated using assembly line balancing logic presented earlier in this paper.
4. Code the specifics of the algorithm: read data, initialize source node, form new arc(s) (with appropriate addressing choice), compute bound, identify terminal node, compute objective function, backtrack for candidate problem-addressed data, copy incumbent solution in LIFO searches, and print the incumbent solution.
5. Validate the complete code.
6. Use the code to solve real problems.

CONCLUSION

The main conclusion is that modularization, as proposed in this paper, can significantly ease the burden of time required to construct branch-and-bound algorithms and thus can contribute to the timeliness of decisions that require branch-and-bound development and/or implementation. Modularization also permits the implementor to test different algorithm alternatives. Theoretically the computation time of code developed using modularization might sometimes be a little slower than tailored code; however, this was not detected in the computational comparisons made. [Received: July 29, 1985. Accepted: September 15, 1986.]

REFERENCES

[1] Balas, E. An additive algorithm for solving linear programs with zero-one variables. *Operations Research*, 1965, *13*, 517-546.

[2] Charlton, J. M., & Death, C. C. A general method for machine scheduling. *International Journal of Production Research*, 1969, 7(3), 207-217.

[3] Dakin, R. J. A tree-search algorithm for mixed integer programming problems. *The Computer Journal*, 1965, *8*(3), 250-255.

[4] Fox, B. L., Lenstra, J. K., Rinnooy Kan, A. H. G., & Schrage, L. E. Branching from the largest upper bound: Folklore and facts. *European Journal of Operational Research*, 1978, *2*, 191-194.
[5] Graves, J. S. On the storage and handling of binary data using FORTRAN with applications to integer programming. *Operations Research*, 1979, *27*, 534-547.
[6] Greenberg, H., & Hegerich, R. L. A branch search algorithm for the knapsack problem. *Management Science*, 1970, *16*, 327-332.
[7] Johnson, R. V. Assembly line balancing algorithms: Computational comparisons. *International Journal of Production Research*, 1981, *19*(3), 277-287.
[8] Johnson, R. V. A branch and bound algorithm for assembly line balancing problems with formulation irregularities. *Management Science*, 1983, *29*, 1309-1324.
[9] Johnson, R. V. Balancing large assembly lines with FABLE. *Management Science*, in press.
[10] Land, A. H., & Doig, A. G. An automatic method of solving discrete programming problems. *Econometrica*, 1960, *28*, 497-520.
[11] Little, J. D. C., Murty, K. G., Sweeney, D. W., & Karel, C. An algorithm for the traveling salesman problem. *Operations Research*, 1966, *11*, 972-989.
[12] Salverson, M. E. The assembly line balancing problem. *Journal of Industrial Engineering*, 1955, *6*(3), 18-25.
[13] Talbot, F. B. Resource-constrained project scheduling with time-resource tradeoffs: The nonpreemptive case. *Management Science*, 1982, *28*, 1197-1210.
[14] Talbot, F. B., & Patterson, J. H. An integer programming algorithm with network cuts for solving the assembly line balancing problem. *Management Science*, 1984, *30*, 85-99.
[15] Talbot, F. B., Patterson, J. H., & Gehrlein, W. V. A comparative evaluation of heuristic line balancing techniques. *Management Science*, 1986, *32*, 430-454.
[16] Wee, T. S., & Magazine, M. J. *An efficient branch and bound algorithm—Part I: Minimize the number of work stations* (Working Paper No. 151). Unpublished manuscript, University of Waterloo, 1981.

Roger V. Johnson is Assistant Professor of Operations Management in the School of Business Administration at the University of Michigan. He earned his Ph.D. in operations management at the University of California-Los Angeles. He has served as Dean of the Faculty of Commerce of Otago University in New Zealand and as Visiting Associate Professor at UCLA. Dr. Johnson has published in *Management Science* and *International Journal of Production Research*. He is a member of Decision Sciences Institute, TIMS, ORSA, and the New Zealand OR Society. His research interests are in assembly line balancing and management, plant and office layout methods, project management, and discrete optimization.