This paper formulates as integer programming problems three methods for assigning data items to registers in the compilation process—the one-one, many-one, and many-few global assignment methods.

Three algorithms are described for obtaining feasible solutions to the many-one and many-few global assignment problems. One provides an optimal solution. The others, which provide good approximations, appear to be sufficiently fast for inclusion in an optimizing compiler.

Compiler assignment of data items to registers

by W. H. E. Day

The assignment of data items to registers is one of the functions performed by a compiler during the preparation of a program for execution. The way in which this function is performed affects the execution characteristics of the program. A good register assignment scheme can, for example, reduce the number of movements of data items between main storage and registers and can allow use of the faster register-to-register type instructions. In this paper, we consider three related methods of assigning data items to registers.

After establishing some basic definitions and assumptions, and distinguishing between global and local assignment, we discuss the three methods of global assignment—one-one, many-one, and many-few.

Global one-one assignment associates exactly one data item with each register in the region of assignment. The method is interesting because it is used in commercially available compilers. In this paper, it is used as a standard for measuring the effectiveness of the many-one and many-few global assignment methods.

Global many-one assignment associates at least one data item with a single register in the region of assignment. We describe a branch-and-bound procedure for obtaining optimal solutions to

this assignment problem and prove that these solutions are indeed optimal. This procedure is used as a standard for measuring the effectiveness of assignment algorithms obtaining possibly nonoptimal solutions.

Global many-few assignment associates at least one data item with each of several registers in the region of assignment. The structure of this problem is identical to that of the global many-one assignment problem, so that the branch-and-bound procedure may be used to obtain optimal solutions to both problems.

Two algorithms are described that obtain possibly nonoptimal solutions to the many-one and many-few global assignment problems. For the problems analyzed, these algorithms seem fast enough to be considered for inclusion in an optimizing compiler. In addition, the solutions generally obtained by them are close to the optimal solution and are significantly better than the corresponding global one-one solution.

Finally, we discuss certain extensions in the use of the global many-few assignment method that may increase the profitability of the final assignments. In particular, ways in which this method may be used to effect the assignment of a data item to more than one register in the region of assignment are considered.

Terminology

In this section we define terms that are relevant to the compiler global assignment of data items to registers. First we describe basic features of a programming language L and consider the structure of a program P written in L. We identify the basic block and the region as structural units of a program and use these terms in distinguishing between local and global assignment. We then define the three global assignment methods: one-one, many-one, and many-few.

programming language

Let L be a programming language. A statement in L is an ordered sequence of delimiters, operators, constants, and identifiers. One subset of identifiers in L contains elements that are used as names of data items. Thus constants and data names are constructs in L that represent data items. Statements in L may be classified as descriptive or as executable, the latter serving to specify operations to be performed on data items.

The occurrence of a constant or data name in an executable statement may be characterized by its effect on the associated data item during statement execution. A data item is *defined* when statement execution causes a new value to become associated with the data item. A data item is *referred to* when the current value of the data

item is required for correct statement execution. Using the semantical rules of L, one may associate with each executable statement an ordered set of constants and data names specifying the temporal sequence of data item definitions and references occurring in statement execution.

program structure

Let P be a program written in L and expressed as a finite ordered set of statements in L:

$$P = \{s_1, s_2, \cdots, s_n\}$$

A basic block B_j is an ordered subset of the statements in P:

$$\begin{aligned} B_j &= \{s_\alpha, s_{\alpha+1}, \cdots, s_\omega\} \\ &= \{s_i \mid s_i \in P, \\ &\qquad s_i \text{ is executed before } s_{i+1}, \\ &\qquad s_i \neq s_\alpha \text{ may not be branched to from } s \in P, \\ &\qquad s_i \neq s_\omega \text{ may not branch to } s \in P\} \end{aligned}$$

P^b is a representation of P as an ordered set of basic blocks:

$$\begin{aligned} P^b &= \{B_1, B_2, \cdots, B_n\} \\ &= \left\{ B_i \;\middle|\; \bigcup_{i=1}^{n} B_i = P, \qquad B_i \cap B_j = \varnothing \qquad \text{for} \quad i \neq j \right\} \end{aligned}$$

P^g is a representation of P as a directed graph in which P^b is the set of vertices and U is the set of directed arcs:

$$P^g = (P^b, U)$$

where:

$$U = \{(x, y) \mid x, y \in P^b, \text{ flow of control may pass from } x \text{ to } y\}$$

A region R_i is a strongly connected subgraph of P^g:

$$R_i = (P_i^b, U_i)$$

where:

$$P_i^b \subseteq P^b$$

$$U_i = \{(x, y) \mid x, y \in P_i^b, \qquad (x, y) \in U\}$$

and there exists a path' joining arbitrary $x, y \in P_i^b$.

P^r is a representation of P as an ordered set of regions:

$$\begin{aligned} P^r &= \{R_1, R_2, \cdots, R_n\} \\ &= \{R_i \mid R_i \neq R_j \qquad \text{for} \quad i \neq j, \\ &\qquad R_i \cap R_j = \varnothing \qquad \text{or} \qquad R_i \subset R_j \qquad \text{for} \quad i < j, \\ &\qquad R_n = P^g\} \end{aligned} \tag{1}$$

A digital computer C performs arithmetic and logical processing of data items using two sets of individually addressable registers:

$G_1^* = \{g_i \mid g_i \text{ is a general register}\}$

$G_2^* = \{g_i \mid g_i \text{ is a floating-point register}\}$

In most instances where an arithmetic or logical operation requires the use of a register from G_j^*, any available $g_i \in G_j^*$ may be assigned.

One of many functions performed in the compilation of P is the assignment of data items to appropriate registers, these assignments being effective during the execution of P on C. Let d represent an element of P, P^b, or P^r in which the assignment of data items to registers is to occur. Define the ordered sets:

$G_j' = \{g_i \mid g_i \in G_j^*,$

g_i is available for assignment everywhere in $d\}$

$N' = \{n_i \mid n_i$ is a data item in P,

n_i may be assigned to registers in $d\}$

types of assignment

The assignment of data items to registers is characterized first in terms of d in the following definitions:

Definition 1: A *local assignment* is a (possibly multiple-valued) mapping of $N \subseteq N'$ onto $G_i \subseteq G_j'$ for $d \in P^b$.

Definition 2: A *global assignment* is a (possibly multiple-valued) mapping of $N \subseteq N'$ onto $G_i \subseteq G_j'$ for $d \in P^r$.

The assignment of data items to registers may be characterized next by the type of mapping that occurs.

Definition 3: A *one-one assignment* is a one-one mapping of $N \subseteq N'$ onto $G_i \subseteq G_j'$. A one-one assignment defines a one-to-one correspondence between N and G_i.

Definition 4: A *many-few assignment* is a single-valued mapping of $N \subseteq N'$ onto $G_i \subseteq G_j'$, where[2] $\mathcal{C}(N) \geq \mathcal{C}(G_i)$.

Definition 5: A *many-one assignment* is a many-few assignment in which $\mathcal{C}(G_i) = 1$.

Definitions 3 and 4 specifically exclude multiple-valued mappings in which a data item is mapped into more than one register. Such mappings may be desirable; this subject is discussed in the section on extensions.

Many-few and many-one assignment methods require a knowledge of the interference characteristics of data items. A data item is *active* at a point in d if it may be referred to subsequent to that point. Two data items *interfere* in d if they are both active at a point in d. *Total interference* exists among data items in a set N if n_i in-

terferes with n_j in d for every $n_i, n_j \in N, i \neq j$. A necessary condition for the assignment of $N \subseteq N'$ to $g \in G_i$ in d is that n_i must not interfere with n_k in d for every $n_i, n_k \in N, i \neq k$.

Global assignment

This section discusses the relevance of the global assignment of data items to registers as a machine-dependent optimization method. It also describes the basic assumptions of the global assignment methods considered in subsequent sections.

The proposed operating environments of a compiler and its compiled programs generally have a significant influence on the compiler design. For example, References 3 and 4 describe compiler designs in which fast compile time is desirable; relatively little importance is attached to the generation of compiled programs having desirable execution characteristics. On the other hand, References 5 and 6 describe compiler designs in which great importance is attached to the generation of compiled programs with desirable execution characteristics, and a reasonable degradation in compile time is tolerated to attain this result. The assignment methods we describe in this paper are particularly relevant to the latter designs, for they attempt to optimize certain of the compiled program's execution characteristics. Specifically we are interested in the compiled program's execution time and length.

Methods of optimizing[5–7] execution characteristics may be classified by the extent of their dependence on the programming language and the digital computer. Optimization methods effecting the assignment of data items to registers are language-independent because high-level programming languages do not usually provide language facilities for manipulating registers. Such methods are machine-dependent because registers are physical components of the digital computer on which the compiled program is to be executed. However, since registers are reasonably standard features of commercially available digital computers, one may claim a degree of machine-independence for register optimization methods not depending on register characteristics unique to a particular digital computer. Some register optimization methods are described in References 6 and 8.

In this paper, we are concerned not only with the physical properties of registers, but also with the properties of the instructions using registers. In executing most arithmetic and logical instructions, an operation is performed using two data items as operands. The first data item must occupy a register and may be replaced by the result of the operation. The second data item may occupy either a register or a main storage location. An instruction is shorter, and its execution is faster, when the second data item occupies a register. It is of particular interest that the result of an operation

is usually left in a register. Thus, when a data item is defined, the new value to be associated with the data item first appears in a register; additional instructions are required if the value is to be preserved in main storage. In addition, the value of a data item may be preserved in a register between consecutive references so long as the register is not otherwise required in the interval. In this case, the register is used as if it were an extension to main storage.

The assignment methods we propose have three important features. First, the assignment of a data item to a register in a region is effective at all points in the region where the data item is active. Second, more than one data item may be assigned to a register. Third, a profit criterion is used to select from among alternative assignments the one more likely to improve the program's execution characteristics. These assignment features, when considered with the previously described register characteristics, enable improvements in program execution characteristics. First, this type of assignment tends to decrease the number of instructions effecting register-register and register-main storage movements of these data items. Second, it tends to increase the number of instances in which both instruction operands occupy registers, thus increasing the use of shorter and faster instructions. Finally, it may be possible to eliminate a data item's main storage location if the data item has been assigned to registers at all points in the program where it is active.

Local and global assignment differ in the extent of the program over which the assignment of data items to registers is effective: local assignment occurs within a basic block, while global assignment occurs within a region. Local assignment is attractive, in part, because a compiler can easily partition a program into basic blocks and derive necessary interference characteristics of data items defined or referred to in the block. Thus, while it may be difficult and time-consuming to assign data items in a region, it is comparatively easy to make an assignment within each basic block in the region. Efficient assignment methods may be developed for local assignment, and may yield optimal register assignments for specific problems. Reference 8, for example, describes an optimal assignment technique for the allocation of index registers.

A weakness in local assignment involves the disposition of data items that are defined or referred to in a block and are active on entry to or exit from the block. Local assignment cannot usually retain assignment history across block boundaries, and so the values of active data items must be moved to main storage for interblock transfers of control. Global one-one assignment offers a partial solution to this problem, for it assigns certain data items to registers, in one-to-one correspondence, throughout the region. Precautions for interblock transfers are unnecessary for these globally assigned data items, although steps now become necessary to preserve the

values of globally assigned data items that are active during inter-region transfers of control. Global one-one assignment does not in itself require knowledge of interference among data items, since it assigns just one data item to a register. The use of the global one-one assignment method in a compiler is described by Reference 6.

A weakness in global one-one assignment is that it is usually incapable of assigning more than one data item to a register in a region. One approach to the solution of this problem is to consider a set of data items for assignment to a register if no two data items in the set interfere at any point in the region. Global many-few assignment attempts to identify this situation and to make, if possible, a more profitable assignment of several data items to the register. This assignment method requires accurate program flow information in the region to calculate the points at which each data item is active and to determine the set of data items with which each data item interferes. This information is used to obtain sets of data items that may be assigned to a register.

A weakness in global many-few assignment is that situations may arise in which precise program flow information is not available: for example, the compiler may be unable to deduce from the program the minimum set of labels to which control passes at a branch statement. To ensure correct execution, the compiler must assume that control may pass at this branch statement to any of a set of labels sufficient to contain the minimum set. The global many-few assignment in this case is usually less profitable than the assignment that would be possible with precise program flow information. In extreme cases, when precise program flow information is unavailable, the resulting global many-few assignment is identical to the corresponding global one-one assignment.

Reference 5 describes an approach to compiler design that is based on the representation of a program as an ordered set of regions (see Equation 1). Optimization methods are applied sequentially to the regions; in the absence of specific information describing the frequency of execution of regions, regions R_i are processed in the index sequence: $\{1, 2, \cdots, n\}$. Now when one program loop is nested within another and the two loops are assigned respectively to R_i and R_j, the method of region identification ensures that $i \leq j$. Thus nested loops are usually optimized before containing loops; to the extent that depth of loop nesting and frequency of execution are related, it also means that more frequently executed regions are optimized before less frequently executed regions. The global assignment methods described in this paper may be considered machine-dependent optimization methods in a compiler having this basic design.

The global assignment methods we describe require a profit to be associated with each data item. This profit measures the improve-

ment in program execution that may occur if the data item is globally assigned to a register in the region being processed. We assume the profit of a particular global assignment to be the sum of the profits of those data items therein assigned to registers. When information is available concerning execution frequencies of blocks in the region, it may be possible to assign data item profits so that the global assignment profit measures with reasonable accuracy the resulting improvement in program execution. When such information is unavailable, it is probably satisfactory to define data item profit to be a linear function of the numbers of definitions of and references to the data item in the region. In this case, the values assigned to the profit equation constants determine whether the profit represents a projected improvement in program size or execution time.

In this paper, we assume that the compiler is able to classify each data item by the register type required to operate on it. Global assignment methods are then used to obtain an assignment of data items to individual registers for each class of registers.

The global assignment methods we describe in this paper assume that the registers available for assignment have uniform characteristics, so that any data item may be assigned to any available register. Violations of this assumption occur in many commercially available computers. In the section on extensions, we indicate how these global assignment methods might be used to assign data items to registers when such special requirements exist.

Global one-one assignment

In this section, we develop a formulation of the global one-one assignment method as an integer programming problem and state an optimal feasible solution to this problem. In this method, after the profit of assigning each data item to a register has been computed, those data items are chosen for assignment to available registers (one data item per register) that maximize profit.

Stated in terms of Definitions 2 and 3, a global one-one assignment is a one-one mapping of $N \subseteq N'$ onto $G_i \subseteq G_j'$ for $d \in P'$.

Now consider the following notation.

m is the number of registers available for assignment in d: $m = \mathcal{C}(G_j')$.

n is the number of data items available for assignment in d: $n = \mathcal{C}(N')$.

$\mathbf{p}$ is a profit vector with dimension $(1 \times n)$: $\mathbf{p} = (p_i)$. p_i is the profit associated with the assignment of $n_i \in N'$ to a register. We adopt the convention that $p_i > 0$ for all $n_i \in N'$.

$\mathbf{x}$ is a data item selection vector with dimension $(n \times 1)$: $\mathbf{x} = [x_i]$. $x_i = 1$ when $n_i \in N'$ is assigned to some $g \in G_j$; otherwise, $x_i = 0$.

z is the objective function, the value of which is to be optimized.

$\mathbf{1}$ is a sum vector of appropriate dimension.

Using this notation, we see that the global one-one assignment method has this formulation:

Maximize	$z = \mathbf{px}$	(2)	**P1**
subject to	$\mathbf{1x} \leq m$	(3)	
where	$x_i \in \{0, 1\}$	(4)	
	$p_i > 0$		

Any $\mathbf{x}$ satisfying Equation 4 is a *solution* to P1. Any $\mathbf{x}$ satisfying Equations 3 and 4 is a *feasible solution* to P1. Any $\mathbf{x}$ satisfying Equations 2-4 is an *optimal feasible solution* to P1.

Theorem 1

Assume (without loss of generality) the elements of N' to be ordered such that $p_i \geq p_j$ for $n_i, n_j \in N'$ and $i < j$. Then

$$\mathbf{x}^* = \{x_j \mid x_j = 1 \quad \text{for} \quad 1 \leq j \leq m,$$
$$x_j = 0 \quad \text{for} \quad m < j \leq n\}$$

is an optimal feasible solution to P1.

The method of indirect proof may be used to prove this theorem.

The ONEONE algorithm denotes a procedure using Theorem 1 to obtain an optimal feasible solution to P1. As an example of its use, let $m = 3$, $n = 9$, and use the profit vector represented by $\mathbf{P}$ in Figure 1. The ONEONE algorithm obtains the optimal feasible solution:

$\mathbf{x} = [1\ 0\ 0\ 0\ 0\ 1\ 0\ 1\ 0]$

for which $z = 269$.

Figure 1 Problem data

C

```
0 1 1 1 0 1 1 1 1
1 0 1 0 0 1 1 0 1
1 1 0 1 1 1 1 1 1
1 0 1 0 1 1 1 1 0
0 0 1 1 0 1 1 1 0
1 1 1 1 1 0 1 1 1
1 1 1 1 1 1 0 1 0
1 0 1 1 1 1 1 0 0
1 1 1 0 0 1 0 0 0
```

P

```
86 24 53 45 75 90 30 93  6
```

Global many-one assignment

In this section we develop a formulation of the global many-one assignment method as an integer programming problem. This problem is to select for assignment (with maximum profit) to a single register in a region a set of data items in which no two data items interfere. We describe two algorithms for obtaining feasible solutions to this problem: one is a branch-and-bound procedure,

and we prove that it obtains an optimal feasible solution; the other obtains a possibly nonoptimal feasible solution. APL functions illustrating these algorithms are presented.

Stated in terms of Definitions 2 and 5, a global many-one assignment is a single-valued mapping of $N \subseteq N'$ onto $g \in G_i'$ for $d \in P^r$, where $\mathcal{C}(N) > 1$.

Now consider the following notation.

$\mathbf{x}$ is a data item selection vector with dimension $(n \times 1)$, where $n = \mathcal{C}(N')$: $\mathbf{x} = [x_i]$. $x_i = 1$ when $n_i \in N'$ is assigned to g; otherwise $x_i = 0$.

$N_i^* \subseteq N'$ is a subset of data items among which there exists total interference in d.

N^{**} is a set of p subsets N_i^* giving a complete description of the data item interference in d: $N^{**} = \{N_1^*, N_2^*, \cdots, N_p^*\}$. It follows that if $n_j, n_k \in N'$ and n_j interferes with n_k in d, then there exists $N_i^* \in N^{**}$ such that: $\{n_j, n_k\} \subseteq N_i^*$.

$\mathbf{A}$ is a data item interference matrix with dimension $(p \times n)$: $\mathbf{A} = (a_{ij})$. $a_{ij} = 1$ if $n_j \in N_i^*$; otherwise $a_{ij} = 0$.

Using this notation, the global many-one assignment method has this formulation:

P2

Maximize	$z = \mathbf{px}$
subject to	$\mathbf{Ax} \leq \mathbf{1}$
	$x_i \in \{0, 1\}$
where	$a_{ij} \in \{0, 1\}$
	$p_j > 0$

P2 is an integer programming problem that might be classified (Reference 9) as a *weighted set matching problem.*

The following terms, originating in Reference 10, are useful in discussing solution techniques for P2. A (*complete*) *solution* S is an assignment of binary values to the data items in N'. S is a particular representation of a solution $\mathbf{x}$ to P2 in which the elements of S are ordered and each element represents a data item and its assigned binary value:

$$S = \{S_j \mid n_{|s_j|} \in N',$$
$$(s_j > 0) \Rightarrow (x_{s_j} = 1),$$
$$(s_j < 0) \Rightarrow (x_{|s_j|} = 0)\}$$

Explicit enumeration is the process of excluding a complete solution from the set of possible optimal feasible solutions to P2. *Implicit enumeration* is the process of excluding a set of complete solutions

T from the set of possible optimal feasible solutions without the explicit enumeration of each $S \in T$. A *partial solution* S^p is an assignment of binary values to the data items in $N \subseteq N'$.

$$S^p = \{s_i \mid n_{|s_i|} \in N,$$
$$(s_i > 0) \Rightarrow (x_{s_i} = 1),$$
$$(s_i < 0) \Rightarrow (x_{|s_i|} = 0)\}$$

A *free variable set* is the set of data items that have not been assigned binary values in S^p.

$$F = \{f \mid n_f \in N', f \notin S^p, -f \notin S^p\}$$

Let S^c be an assignment of binary values to all n_f, $f \in F$.

$$S^c = \{s \mid |s| \in F,$$
$$(s > 0) \Rightarrow (x_s = 1),$$
$$(s < 0) \Rightarrow (x_{|s|} = 0)\}$$

Then a *completion* of S^p is a complete solution determined by S^p and S^c.

Let z' be the objective function value of the most profitable feasible solution to P2 yet obtained. To *fathom* S^p is to determine that among all completions of S^p either there exists no feasible completion more profitable than z', or there exists a distinct most profitable completion, with objective function value z'', such that $z'' > z'$. The set of completions of a fathomed partial solution is implicitly enumerated.

Branch-and-bound procedures[11,12] form a distinctive subclass of those enumerative methods obtaining optimal feasible solutions to problems like P2. These procedures use a branching procedure and a set of bounding rules to obtain the complete implicit enumeration of solutions to the problem. The branching procedure generates for analysis an ordered sequence of partial solutions and terminates only when all solutions to the problem have been implicitly enumerated. The set of bounding rules establishes a lower bound on the value of the optimal objective function and an upper bound on the objective function value of the most profitable feasible completion of a given partial solution. These bounds then determine if the completions of the given partial solution may be implicitly enumerated.

The OPTIMAL algorithm (Table 1) is a branch-and-bound procedure that obtains an optimal feasible solution to P2. It has three basic components: a procedure obtaining an initial approximation to the optimal feasible solution (Table 1, step 2); a fathoming procedure (step 3); and a branching procedure (steps 4 and 5). The final feasible solution is optimal if one can prove that:

Table 1 The OPTIMAL algorithm

1. Set: $S = \varnothing$
   $i = 0$
2. Obtain a complete feasible solution S' with objective function value z' that approximates the optimal feasible solution.
3. Try to fathom the partial solution S. If the process determines any complete feasible solution S'' with corresponding z'' such that $z'' > z'$, then set:
   $z' = z''$
   $S' = S''$
4. If the attempt to fathom $S = \{s_1, \cdots, s_i\}$ fails, then use the free variable set F associated with S to set:
   $s_{i+1} = j \in F$
   $i = i + 1$
   where this expanded S is a feasible partial solution to P2. Go to step 3.
5. Otherwise S is fathomed. If $i > 0$ and there exists $k \leq i$ for which $s_k > 0$, locate the largest integer k for which $s_k > 0$. Set:
   $S = \{s_1, \cdots, s_k\}$
   $s_k = -s_k$
   $i = k$
   and go to step 3.
6. Otherwise the algorithm terminates with the optimal feasible solution to P2, denoted by S^* and z^*:
   $S^* = S'$
   $z^* = z'$

- The branching procedure terminates only when all complete solutions to P2 have been implicitly enumerated.
- The bounding rules used to fathom a partial solution are valid.

In addition, the performance of the OPTIMAL algorithm is improved if:

- The branching procedure generates a sequence of fathomed partial solutions in which each complete solution to P2 occurs as the completion of exactly one fathomed partial solution.
- Step 2 obtains a feasible solution with a profit that is close to that of the optimal feasible solution.

These issues are addressed in the following sections. First we describe the algorithm used in step 2 and give an implementation of it as an APL function. (The effectiveness with which this algorithm approximates the profit of the optimal feasible solution is discussed in the section on test results.) We next prove the two essential features of the branching procedure. Then we describe the bounding rules used to fathom a partial solution and prove their validity.

the ESTIMATE algorithm

One may visualize the output of the OPTIMAL algorithm as an ordered sequence of ordered pairs:

$$\sum = \{(S_1', z_1'), \cdots, (S_p', z_p')\}$$

where S_i' is a complete feasible solution to P2, and z_i' is the objective function value of S_i'. This sequence converges to an optimal feasible solution, denoted by (S^*, z^*), so that

$$(S_p', z_p') = (S^*, z^*)$$

During its execution, the OPTIMAL algorithm uses the current most profitable feasible solution, (S_i', z_i'), in the fathoming process. Usually when $z_i' \ll z^*$, the fathoming process is ineffective in implicitly enumerating large sets of complete solutions. Step 2, hereafter called the ESTIMATE algorithm, attempts to avoid this situation by generating a feasible solution with an objective function value approximating that of the optimal feasible solution. This initial feasible solution becomes (S_1', z_1'), the first element in $\sum$.

Now consider the following notation. The data item interference matrix **A** is a particular representation of the interference existing among data items $n_i \in N'$ in $d \in P^r$. The data item interference matrix **C** is an alternative representation of this interference information:

$$\mathbf{C} = (c_{ij}) = (\mathbf{c}_1, \mathbf{c}_2, \cdots, \mathbf{c}_n)$$

C is a symmetric matrix with order $n = \mathcal{C}(N')$. $c_{ij} = c_{ji} = 1$ when $n_i, n_j \in N'$, $i \neq j$, and n_i interferes with n_j in d; otherwise $c_{ij} = 0$. $\mathbf{c}_i$ is the ith column vector in **C** and specifies the interference in d between n_i and each $n_j \in N'$, $j \neq i$. **E** is a matrix of order n describing n complete feasible solutions to P2:

$$\mathbf{E} = (e_{ij}) = (\mathbf{e}_1, \mathbf{e}_2, \cdots, \mathbf{e}_n)$$

$\mathbf{e}_i$ is the ith column vector in **E** and describes the ith complete feasible solution to P2. $e_{ji} = 1$ when $n_j \in N'$ is assigned to the register in the ith complete solution; otherwise $e_{ji} = 0$. From $\mathbf{e}_i$, one may obtain the complete solution S_i, with objective function value z_i, by:

$$z_i = \mathbf{pe}_i$$

$$S_i = \{s \mid n_{|s|} \in N',$$
$$(e_{ji} = 1) \Rightarrow (j \in S_i),$$
$$(e_{ji} = 0) \Rightarrow (-j \in S_i)\}$$

The ESTIMATE algorithm generates $n = \mathcal{C}(N')$ complete feasible solutions, selecting the most profitable to become $(S_1', z_1') \in \sum$:

$$z_1' = \mathbf{pe}_k = \max_i (\mathbf{pe}_i) \tag{5}$$

$$S_1' = S_k \tag{6}$$

Two features of the ESTIMATE algorithm determine the method by which the n complete feasible solutions are constructed. First,

Table 2 The ESTIMATE algorithm

1. Assign $n_i \in N'$ to the register in the partial solution represented by $\mathbf{e}_i$. This is accomplished by setting:*

   $\mathbf{E} = \sim\mathbf{C}$

2. Calculate the sequence Q using Equations 7 and 9. Then execute step 3 once for each $q_j \in Q$ in the index sequence:

   $\{n, n-1, \cdots, 1\}$

3. Assign n_{q_j} to the register in every partial solution by calculating:

   $\mathbf{e}_i = \mathbf{e}_i \wedge (\sim\mathbf{c}_{q_j})$

   for each

   $i \in \{k \mid e_{q_j k} = 1\}$

4. Each $\mathbf{e}_i$ now represents a complete feasible solution. Calculate (S_1', z_1') using Equations 5 and 6.

* $\sim\mathbf{C}$ is the complement of C, in which $\sim c_{ij} = 1 - c_{ij}$

n_i is always the first data item assigned in the ith partial solution. Second, a particular sequence, Q, determines the order in which data items subsequent to the first are selected for assignment in each partial solution.

$$Q = \{q_i \mid n_{q_i} \in N', (r_{q_i} \leq v_{q_i}) \Rightarrow (i < j)\} \tag{7}$$

where:

$$r_i = \frac{(n - 1\mathbf{c}_i)}{n} \cdot \frac{p_i}{\max\limits_j (p_j)} \tag{8}$$

In this definition, r_i is a dimensionless quantity that we interpret as an indicator of the potential for obtaining a profitable complete feasible solution when n_i is assigned in a partial feasible solution. The first term in r_i is large if n_i interferes with few other data items; the second term is large when the profit of n_i approximates that of the most profitable data item. It follows that the indices of profitable data items interfering with few other data items should tend to occur late in the sequence Q. The denominator of Equation 8 is constant and does not affect the relative placement of data item indices in Q. Thus one may construct Q using:

$$r_i = (n - 1\mathbf{c}_i)p_i \tag{9}$$

Table 2 describes the essential features of the ESTIMATE algorithm; these steps illustrate the transformation of $\mathbf{E}$ from an initial value (step 1) to a final value (step 4). At any given instant in the iterative execution of step 3, $\{x \mid e_{xi} = 0\}$ is the set of data items that will not be assigned to the register, while $\{x \mid e_{xi} = 1\}$ is the set of data items assigned, or still being considered for assignment, to the register in the final complete feasible solution represented by $\mathbf{e}_i$. When control passes to step 4, all outstanding data item candi-

Figure 2 The ESTIMATE APL function

```
      ∇ESTIMATE[⎕]∇
     ∇ P ESTIMATE C;E;J;Q;X
[1]    J←(ρE←~C)[1]
[2]    Q←⍋P×+/~C
[3]   LA:E[;X]←E[;X]∧⍉((ρX←E[Q[J];]/ιρP),ρP)ρ~C[;Q[J]]
[4]    →LA×ι0<J←J-1
[5]    Z←⌈/P+.×E
[6]    S←(ιρP)×1-2×~E[;(P+.×E)ιZ]
     ∇
```

Figure 3 Example using the ESTIMATE APL function

```
      P ESTIMATE C

      Z
161
      S
1  ¯2  ¯3  ¯4  5  ¯6  ¯7  ¯8  ¯9

      A  INITIAL VALUE OF E:

1 0 0 0 1 0 0 0 0
0 1 0 1 1 0 0 1 0
0 0 1 0 0 0 0 0 0
0 1 0 1 0 0 0 0 1
1 1 0 0 1 0 0 0 1
0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0 1
0 1 0 0 0 0 0 1 1
0 0 0 1 1 0 1 1 1

      A  FINAL VALUE OF E:

1 0 0 0 1 0 0 0 0
0 1 0 1 0 0 0 1 0
0 0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0
1 1 0 0 1 0 0 0 1
0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 1 0 1

      A  VALUE OF Q:

9 3 7 6 2 4 1 8 5
```

dates have been considered for assignment in each solution; each $\mathbf{e}_i$ represents at this point a complete feasible solution.

Figure 2 shows an implementation of ESTIMATE. The correspondence between symbols used in the text and APL variables is: $\mathbf{C} \leftrightarrow C$, $\mathbf{p} \leftrightarrow P$, $\mathbf{E} \leftrightarrow E$, $Q \leftrightarrow Q$, $z_1' \leftrightarrow Z$, and $S_1' \leftrightarrow S$. P and C are passed to ESTIMATE as arguments, while S and Z are returned as global variables.

Figure 3 shows the application of ESTIMATE to the problem described in Figure 1. The final values of Z and S represent an optimal feasible solution. We also show the value of Q and the initial and final values of E calculated by ESTIMATE for this problem.

the branching procedure

We now state two essential features of the OPTIMAL algorithm branching procedure. Their proofs are in Appendices A and B and stem from an analysis in Reference 10 of a problem more general than P2.

The following notation and terms are used in the discussion. $\sum^f$ is the sequence of fathomed partial solutions generated by the OPTIMAL algorithm branching procedure.

$$\sum^f = \{S_1^p, \cdots, S_w^p\}$$

A sequence of fathomed partial solutions is *nonredundant* if each complete solution fathomed in the sequence occurs as the completion of exactly one fathomed partial solution. $S_i^p \in \sum^f$ is nonredundant if none of its completions are completions of any $S_j^p \in \sum^f$, $j \neq i$. A sufficient condition for $S_i^p \in \sum^f$ to be nonredundant is that it include the complement of an element $s \in S_j^p$ for each $S_j^p \in \sum^f$, $j \neq i$.

Theorem 2

$\sum^f$, the sequence of fathomed partial solutions generated by the OPTIMAL algorithm branching procedure, is nonredundant. The proof of this theorem is in Appendix A.

Theorem 3

$\sum^f$, the sequence of fathomed partial solutions generated by the OPTIMAL algorithm branching procedure, terminates when all 2^n complete solutions to P2 have been implicitly enumerated. The proof of this theorem is in Appendix B.

the fathoming procedure

Consider the following notation.

F^f is a subset of F in which each element is a candidate for assignment in a feasible completion of S^p: $F^f = \{f_i \mid f_i \in F, s \in S^p, s > 0, c_{f_i s} = 0\}$.

$\mathbf{C}^*$ is a symmetric submatrix of $\mathbf{C}$, with order $\mathcal{C}(F^f)$, describing the interference among the data items $n_f, f_i \in F^f$: $\mathbf{C}^* = (c^*_{ij})$. $c^*_{ij} = 1$ when $c_{f_i f_j} = 1$ and $f_i, f_j \in F^f$; otherwise $c^*_{ij} = 0$.

Theorems 4 and 5 identify two conditions in which it is possible to determine an optimal feasible completion of a partial solution S^p.

Theorem 4 If $F^f = \varnothing$ then

$$S^c = \{s \mid |s| \in F, s < 0\}$$

determines S'', an optimal feasible completion of S^p, with objective function value z''.

Proof: We use the method of indirect proof to obtain a contradiction. Suppose there exists T^c determining T, a feasible completion of S^p, with objective function value $z > z''$. Since $p_i > 0$ for $n_i \in N'$, T cannot be more profitable than S'' unless there exists some $t \in T^c$, $t > 0$. But since T is a feasible completion of S^p, we must have $t \in F^f$. This contradicts the hypothesis that $F^f = \varnothing$ and proves the theorem.

Theorem 5 If $F^f \neq \varnothing$ and $\mathbf{C}^* = \mathbf{0}$ then

$$S^c = \{s \mid |s| \in F, (s \in F^f) \Rightarrow (s > 0), (|s| \notin F^f) \Rightarrow (s < 0)\}$$

determines S'', an optimal feasible completion of S^p, with objective function value z''.

Proof: We use the method of indirect proof to obtain a contradiction. Suppose there exists T^c determining T, a feasible completion of S^p, with objective function value $z > z''$. Since $p_i > 0$ for $n_i \in N''$, T cannot be more profitable than S'' unless there exists some $t \in T^c$, $t > 0$ where $t \notin S^c$. But since T is a feasible completion of S^p, we must have $t \in F^f$. However, if $t \in F^f$, then $t \in S^c$ by the method of constructing S^c. This contradicts the requirement that $t \notin S^c$ and proves the theorem.

Theorem 6 proves that it is possible to fathom a partial solution when it satisfies the conditions of Theorem 4 or Theorem 5.

Theorem 6 If Theorem 4 or Theorem 5 determines an optimal feasible completion of S^p, then S^p is fathomed.

Proof: Let S'', with objective function z'', be the optimal feasible completion of S^p determined by Theorem 4 or Theorem 5. Let z'

be the objective function value of the most profitable feasible solution to P2 yet obtained. By definition, S^p is fathomed when there exists either no completion of S^p more profitable than z' or a distinct most profitable completion of S^p that is more profitable than z'. Either $z'' > z'$ or $z'' \leq z'$. If $z'' > z'$, then S'' satisfies the second condition and S^p is fathomed. If $z'' \leq z'$, S'' ensures that the first condition is satisfied, since no feasible completion of S^p can be more profitable than it. Thus S^p is fathomed, and the theorem is proved.

There exist criteria by which one may determine that certain n_i, $i \in F^f$ cannot be assigned in completions of S^p more profitable than the most profitable feasible solution yet obtained. Such data items may be deleted immediately from the F^f associated with S^p. In turn, these deletions may hasten the fathoming of S^p addressed by Theorem 6. Theorem 7 states a basic criterion for the deletion of data items from F^f.

Theorem 7

z' is the objective function value of the most profitable feasible solution to P2 yet obtained. n_i, $i \in F^f$, is a candidate for assignment in a completion of S^p, and has associated with it:

$$U_i = \{j \mid j \in F^f, c_{ij} = 0\}$$

If

$$\sum_{j \in U_i} p_j \leq z' - \sum_{\substack{j \in S^p, \\ j>0}} p_j$$

then n_i cannot be assigned in completions of S^p more profitable than z'.

Proof: We prove the contrapositive, that if S^c determines a completion of S^p more profitable than z' and $i \in S^c$, $i > 0$, then:

$$\sum_{j \in U_i} p_j > z' - \sum_{\substack{j \in S^p, \\ j>0}} p_j \tag{10}$$

Since S^c determines a completion of S^p more profitable than z', we have:

$$\sum_{j \in U} p_j > z' - \sum_{\substack{j \in S^p, \\ j>0}} p_j \tag{11}$$

where:

$$U = \{j \mid j \in S^c, j > 0\}$$

Since S^c determines a feasible completion of S^p, we have:

$$c_{ij} = 0 \quad \text{for all} \quad j \in U$$

From this and the definition of U_i, it follows that $U \subseteq U_i$. Since $p_j > 0$ for $n_j \in N'$, it then follows that:

$$\sum_{j \in U} p_j \leq \sum_{j \in U_i} p_j$$

Substituting this result in Equation 11 yields Equation 10, which we set out to prove.

Theorems 8 and 9 describe two characteristics exhibited by completions of S^p. Theorem 8 states an upper bound on the maximum number of data items that may be assigned in a completion of S^p. This may be used in the construction of rules governing the removal of data items from F^f. Theorem 9 states an upper bound on the minimum number of separate feasible completions of S^p necessary to include the assignment of every $f \in F^f$. This theorem is of particular relevance in the consideration of many-few assignment techniques: if $S^p = \varnothing$ and $F^f = N'$, then Theorem 9 states an upper bound on $\mathcal{C}(G_j)$, the number of registers required as images in a single-valued mapping of N' onto $G_j \subseteq G_j'$.

Theorem 8 T is a set, with maximum cardinality, of data items $f \in F^f$, all of which may be assigned in a single completion of S^p. Assume the elements of F^f to be ordered such that if $m = \mathcal{C}(F^f)$, then:

$$\sum_{k=1}^{m} c_{ik}^* \leq \sum_{k=1}^{m} c_{jk}^*$$

for $f_i, f_j \in F^f$ and $i < j$.

Then:

$$D = \max_{i=1,\cdots,m} \left\{ i - \sum_{j=1}^{i} c_{ij}^* \right\} \geq \mathcal{C}(T) \tag{12}$$

The proof of this theorem is in Appendix C.

Theorem 9 T is a set, with minimum cardinality, of elements $S_i^c \in T$ such that: each S_i^c determines a feasible completion of a given S^p; and each $f \in F^f$ is assigned in exactly one $S_i^c \in T$. Assume the elements of F^f to be ordered such that if $m = \mathcal{C}(F^f)$, then:

$$\sum_{k=1}^{m} c_{ik}^* \leq \sum_{k=1}^{m} c_{jk}^*$$

for $f_i, f_j \in F^f$ and $i < j$.

Then:

$$R = 1 + \max_{i=1,\cdots,m} \left\{ \sum_{j=i}^{m} c_{ij}^* \right\} \geq \mathcal{C}(T) \tag{13}$$

The proof of this theorem is in Appendix D.

Figure 4 shows two APL functions, D and R, which calculate the upper bounds specified by Theorem 8 and Theorem 9. The APL variable C corresponds to $\mathbf{C}^*$ as defined in the text. C is passed to each APL function as an argument, while the value of the upper bound is returned as an explicit result. Given C as defined in Figure 1, the upper bounds obtained by D and R have the values 3 and 6,

Figure 4 The D and R APL functions

```
          ∇D[⎕]∇
        ∇ Z←D C
[1]       Z←⌈/Z-+/(Z∘.>Z←⍳(⍴C)[1])∧C[⍋+/C;⍋+/C]
        ∇

          ∇R[⎕]∇
        ∇ Z←R C
[1]       Z←1+⌈/+/(Z∘.<Z←⍳(⍴C)[1])∧C[⍋+/C;⍋+/C]
        ∇
```

Figure 5 The OPTIMAL APL function

```
          ∇OPTIMAL[⎕]∇
      ∇ P OPTIMAL C;F;E;S;SC;T
[1]   STEP1:S←⍳0
[2]   STEP2:P ESTIMATE C
[3]   STEP3:F←E←~(⍳⍴P)∈⍳S
[4]    E[(∨/[1] C[(S>0)/S;])/⍳⍴P]←0
[5]    T←+/E
[6]   THEOREM7:E[((Z-+/P⌈(S>0)/S])≥(E×P)+.×~C)/⍳⍴P]←0
[7]    →(~2+I26)×⍳T≠+/E
[8]    SC←⍳0
[9]    →(2+I26)×⍳0≠+/E
[10]  THEOREM4:SC←-F/⍳⍴P
[11]   →(2+I26)×⍳(0=+/E)∨0≠+/+/C[E/⍳⍴P;E/⍳⍴P]
[12]  THEOREM5:SC←(E/⍳⍴P),-(P-E)/⍳⍴P
[13]   →(2+I26)×⍳Z>+/P[(0<S,SC)/S,SC]
[14]   Z←+/P[(S>0)/S←(S,SC)[⍋⍳S,SC]]
[15]   →STEP5×⍳0≠⍴SC
[16]  STEP4:S←S,(∨E×P×+/~C)[1]
[17]   →STEP3
[18]  STEP5:S←(1-(⌽×S)⍳1)↓S
[19]  STEP6:→0×⍳0=⍴S
[20]   S[⍴S]←-S[⍴S]
[21]   →STEP3
      ∇
```

respectively. For this problem, the least upper bounds have the values 2 and 6, respectively.

implementation of OPTIMAL

Figure 5 shows an implementation of the OPTIMAL algorithm as an APL function. The correspondence between symbols used in the text and APL variables is: $\mathbf{C} \leftrightarrow C$, $\mathbf{p} \leftrightarrow P$, $S \leftrightarrow \underline{S}$, $S^c \leftrightarrow SC$, $F \leftrightarrow F$, $F^f \leftrightarrow \underline{F}$, $z' \leftrightarrow Z$, and $S' \leftrightarrow S$. P and C are passed to the OPTIMAL function as arguments, while S and Z are returned as global variables. APL statement labels identify previously described theorems in the fathoming procedure and show the correspondence between the function statements and the OPTIMAL algorithm steps in Table 1.

Our experience using the OPTIMAL algorithm for the solution of nontrivial problems indicates that execution of the ESTIMATE algorithm accounts for a negligible fraction of the total execution time. We have also used the ESTIMATE algorithm, by itself, for the solution of nontrivial problems. This approach is faster, but it obtains a possibly nonoptimal solution.

Global many-few assignment

In the section on global many-one assignment, we developed a formulation of that assignment method. This may take the general form:

$$\text{Maximize} \qquad z = \mathbf{p}^*\mathbf{x}^* \tag{14}$$

$$\left.\begin{array}{ll}\text{subject to} & \mathbf{A}^*\mathbf{x}^* \leq \mathbf{1} \\ & x_i^* \in \{0, 1\}\end{array}\right\} \tag{15}$$

$$\left.\begin{array}{ll}\text{where} & a_{ij}^* \in \{0, 1\} \\ & p_j^* > 0\end{array}\right\} \tag{16}$$

in which we substitute:

$$\mathbf{p}^* = \mathbf{p}$$

$$\mathbf{A}^* = \mathbf{A}$$

$$\mathbf{x}^* = \mathbf{x}$$

We also described the OPTIMAL algorithm for obtaining an optimal feasible solution to P2 and the ESTIMATE algorithm for obtaining a possibly nonoptimal feasible solution.

In this section, we develop a formulation of the global many-few assignment method. Its structure is identical to that of P2, so that the algorithms obtaining feasible solutions to P2 may also be used for the global many-few assignment problem. The many-one assignment problem is extended to assign multiple registers simultaneously. Additional constraints are introduced to limit the assignment of each data item to no more than one register. We describe three algorithms for obtaining feasible solutions to this problem: one obtains an optimal feasible solution; the other two obtain possibly nonoptimal feasible solutions. APL functions illustrating these three algorithms are presented.

Stated in terms of Definitions 2 and 4, a global many-few assignment is a single-valued mapping of $N \subseteq N'$ onto $G_i \subseteq G_i'$ for $d \in P^r$, where $\mathcal{C}(N) \geq \mathcal{C}(G_i)$.

Now consider the following notation.

m is the number of registers available for assignment in d: $m = \mathcal{C}(G_i')$.

n is the number of data items available for assignment in d: $n = \mathcal{C}(N')$.

$\mathbf{p}^{(j)}$, $\mathbf{A}^{(j)}$, and $\mathbf{C}^{(j)}$ represent the jth replications of $\mathbf{p}$, $\mathbf{A}$, and $\mathbf{C}$, respectively.

$\mathbf{x}_k$ is a data item selection vector with dimension $(n \times 1)$: $\mathbf{x}_k = [x_{ik}]$. $x_{ik} = 1$ when $n_i \in N'$ is assigned to $g_k \in G_j$; otherwise $x_{ik} = 0$.

If we consider g_k to be the single register in a global many-one assignment problem, any feasible solution must satisfy the constraints:

$$\mathbf{A}\mathbf{x}_k \leq \mathbf{1}$$

These constraints must also be satisfied when we consider g_k to be a register in a global many-few assignment problem. Thus, a partial specification of the constraints for this latter problem is:

$$\begin{bmatrix} \mathbf{A}^{(1)} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{A}^{(2)} & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \\ \mathbf{0} & \mathbf{0} & & \mathbf{A}^{(m)} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_m \end{bmatrix} \leq \mathbf{1} \tag{17}$$

Since global many-few assignment is defined to be a single-valued mapping, additional constraints are required to limit the assignment of each data item to at most one register. Such constraints have the form:

$$\sum_{i=1}^{m} \mathbf{I}\mathbf{x}_i \leq \mathbf{1} \tag{18}$$

where I is the identity matrix. Merging Equation 18 with Equation 17, we obtain this formulation of the global many-few assignment method:

P3

$$\text{Maximize} \quad z = (\mathbf{p}^{(1)}\mathbf{p}^{(2)} \cdots \mathbf{p}^{(m)}) \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_m \end{bmatrix}$$

$$\text{subject to} \quad \begin{bmatrix} \mathbf{I}^{(1)} & \mathbf{I}^{(2)} & \cdots & \mathbf{I}^{(m)} \\ \mathbf{A}^{(1)} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{A}^{(2)} & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \\ \mathbf{0} & \mathbf{0} & & \mathbf{A}^{(m)} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_m \end{bmatrix} \leq \mathbf{1}$$

where $a_{ij} \in \{0, 1\}$

$x_{ij} \in \{0, 1\}$

$p_i > 0$

P3 has the structure of P2 shown by Equations 14–16, where:

Figure 6 The OPTSOL APL function

```
      ∇OPTSOL[⎕]∇
    ∇ OPTSOL M;L;N;S;Z
[1]   N←⍴P
[2]   L←M×N
[3]   (L⍴P) OPTIMAL(L,L)⍴(1 3 2 4)⍉(M,M,N,N)⍴(,C),(L×N)⍴(⍳N)∘.=⍳N
[4]   'PROFIT:  ';Z
[5]   'SOLUTION:  ';(M,N)⍴(L⍴⍳⍴P)××S
    ∇
```

$$\mathbf{p}^* = (\mathbf{p}^{(1)}\mathbf{p}^{(2)} \cdots \mathbf{p}^{(m)}) \tag{19}$$

$$\mathbf{A}^* = \begin{bmatrix} \mathbf{I}^{(1)} & \mathbf{I}^{(2)} & \cdots & \mathbf{I}^{(m)} \\ \mathbf{A}^{(1)} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{A}^{(2)} & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \\ \mathbf{0} & \mathbf{0} & & \mathbf{A}^{(m)} \end{bmatrix} \tag{20}$$

$$\mathbf{x}^* = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_m \end{bmatrix}$$

Associated with $\mathbf{A}^*$, as defined by Equation 20, is the corresponding data item interference matrix $\mathbf{C}^*$:

$$\mathbf{C}^* = \begin{bmatrix} \mathbf{C}^{(1)} & \mathbf{I} & \cdots & \mathbf{I} \\ \mathbf{I} & \mathbf{C}^{(2)} & \cdots & \mathbf{I} \\ \vdots & \vdots & \ddots & \\ \mathbf{I} & \mathbf{I} & & \mathbf{C}^{(m)} \end{bmatrix} \tag{21}$$

The following algorithm may be used to obtain feasible solutions to P3. Assume the input to be **p**, **A**, and m. Derive **C** from **A**. Construct $\mathbf{p}^*$ and $\mathbf{C}^*$, as defined by Equations 19 and 21, and pass them as arguments to the OPTIMAL or the ESTIMATE algorithm. Either algorithm returns a feasible solution described by z' and S'; the data item assignments to each of the m registers are easily derived from S'.

Figure 7 Example using the OPTSOL APL function

```
      OPTSOL 1
PROFIT:  161
SOLUTION:
  1 ¯2 ¯3 ¯4  5 ¯6 ¯7 ¯8 ¯9

      OPTSOL 2
PROFIT:  278
SOLUTION:
  1 ¯2 ¯3 ¯4  5 ¯6 ¯7 ¯8 ¯9
 ¯1  2 ¯3 ¯4 ¯5 ¯6 ¯7  8 ¯9

      OPTSOL 3
PROFIT:  368
SOLUTION:
  1 ¯2 ¯3 ¯4  5 ¯6 ¯7 ¯8 ¯9
 ¯1 ¯2 ¯3 ¯4 ¯5  6 ¯7 ¯8 ¯9
 ¯1  2 ¯3 ¯4 ¯5 ¯6 ¯7  8 ¯9
```

OPTSOL (Figure 6) obtains an optimal feasible solution to P3 in the manner described above. The correspondence between symbols used in the text and APL variables is: $\mathbf{C} \leftrightarrow C$, $\mathbf{p} \leftrightarrow P$, $m \leftrightarrow M$, $z' \leftrightarrow Z$, and $S' \leftrightarrow S$. M is passed to OPTSOL as an argument, whereas P and C are considered global variables defined external to it. Statements [1] and [2] provide notational conveniences. Statement [3] constructs $\mathbf{p}^*$ and $\mathbf{C}^*$ and invokes the OPTIMAL function. Statements [4] and

Figure 8 The ESTSOL1 APL function

```
      ∇ESTSOL1[⎕]∇
    ∇ ESTSOL1 M;L;N;S;Z
[1]   N←⍴P
[2]   L←M×N
[3]   (L⍴P) ESTIMATE(L,L)⍴(1 3 2 4)⍉(M,M,N,N)⍴(,C),(L×N)⍴(⍳N)∘.=⍳N
[4]   'PROFIT:  ';Z
[5]   'SOLUTION:  ';(M,N)⍴(L⍴⍳⍴P)××S
    ∇
```

[5] print a description of the optimal feasible solution. Statement [4] prints the optimal objective function value. The kth row printed by statement [5] represents an assignment S_k of $N \subseteq N'$ to some $g_k \in G_j$, where:

$$S_k = \{s \mid n_{|s|} \in N',$$

$$(s > 0) \Rightarrow (n_s \quad \text{is assigned to} \quad g_k),$$

$$(s < 0) \Rightarrow (n_{|s|} \quad \text{is not assigned to} \quad g_k)\}$$

Figure 7 shows the application of OPTSOL to the problem described in Figure 1, for various values of m.

ESTSOL1 (Figure 8) is an APL function obtaining a possibly nonoptimal feasible solution to P3. ESTSOL1 is identical to OPTSOL except that the ESTIMATE function is invoked instead of the OPTIMAL function. The application of ESTSOL1 to the problem described in Figure 1 yields results equivalent to those shown in Figure 7.

An alternative algorithm is to consider the global many-few assignment problem as a sequence of m global many-one assignment problems. In this approach, the ESTIMATE algorithm is invoked to obtain an assignment of data items to a single register. Assigned data items are immediately deleted from the problem to prevent their subsequent assignment to other registers. The ESTIMATE algorithm is invoked m times, or until no data items remain to be assigned to registers.

ESTSOL2 (Figure 9) obtains a possibly nonoptimal feasible solution to P3 in the manner described above. The correspondence between symbols used in the text and APL variables is: $\mathbf{C} \leftrightarrow C$, $\mathbf{p} \leftrightarrow P$, $m \leftrightarrow M$, $z' \leftrightarrow Z$, and $S' \leftrightarrow S$. M is passed to ESTSOL2 as an argument, while P and C are considered global variables defined external to it. The APL variable $\underline{S}$ describes the incomplete feasible solution as it is determined by successive invocations of the ESTIMATE function. The APL variable T deletes assigned data items from the problem. Statements [3] and [5] bound the loop obtaining the sequence of solutions to the global many-one assignment problem. Statement [3] invokes the ESTIMATE function. Statement [4] updates the incomplete solution with the partial solution just obtained.

Figure 9 The ESTSOL2 APL function

```
      ∇ESTSOL2[⎕]∇
    ∇ ESTSOL2 M;S;S;T;Z
[1]   S←⍳0
[2]   T←(⍴P)⍴1
[3]   LA:(T/P) ESTIMATE T/T←C
[4]   S←S,(⍳⍴P)×1-2×~T\0<S
[5]   →LA×⍳(0<N←N-1)∧0≠+/T←T≠T\0<S
[6]   'PROFIT:  ';+/P[(0<S)/S]
[7]   'SOLUTION:  ';(((⍴S)⍴(⍴P)),⍴P)⍴S
    ∇
```

Statement [5] corrects the specification of data items already assigned in the problem; it also closes the loop when data items and registers remain to be assigned. Statements [6] and [7] print a description of the final feasible solution in the format used by the OPTSOL and ESTSOL1 functions.

The application of ESTSOL2 to the problem described in Figure 1 yields results equivalent to those shown in Figure 7.

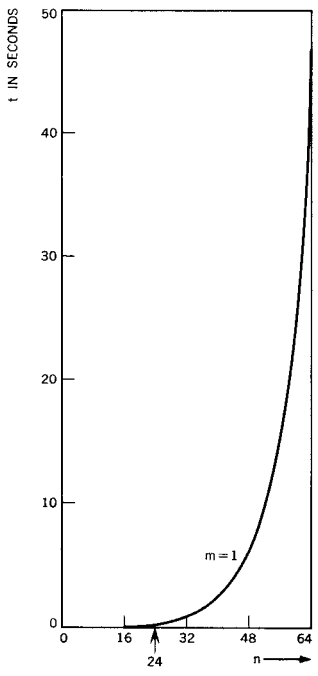
Although OPTSOL, ESTSOL1, and ESTSOL2 obtain feasible solutions to P3 in the general case, they obtain feasible solutions to the global one-one and many-one assignment problems as special cases.

When total interference exists among all data items being considered for assignment, **A** is a matrix with dimension $(1 \times n)$ in which $a_{1i} = 1, 1 \leq i \leq n$. The corresponding data item interference matrix is: $\mathbf{C} = \sim\mathbf{I}$. In this case, OPTSOL, ESTSOL1, and ESTSOL2 obtain an optimal feasible solution to P3 that is equivalent to the ONEONE optimal feasible solution to P1.

When $m = 1$, OPTSOL obtains an optimal feasible solution to P2. When $m = 1$, ESTSOL1 and ESTSOL2 obtain the same result, a possibly nonoptimal feasible solution to P2.

Test results

Figure 10 OPTIMAL average execution times required by OPTSOL



In this section, we use results of the ONEONE and OPTSOL algorithms as standards in comparing the execution characteristics of the ESTSOL1 and ESTSOL2 algorithms. These four algorithms were used to obtain solutions to 400 assignment problems generated for use in the comparison. The ESTSOL1, ESTSOL2, and OPTSOL problem solutions were obtained by executing programs on an IBM System/360 Model 65. Execution time required to obtain the final feasible solution, and final feasible solution profit, are the criteria used to measure the execution characteristics of these algorithms.

The APL functions in this paper are documentation and were *not* used with the APL\360 System to obtain the results reported here. Instead, versions of ESTSOL1, ESTSOL2, OPTSOL, ESTIMATE, and OPTIMAL were written in an experimental programming language and were then compiled into IBM System/360 Operating System assembler language programs. The ONEONE algorithm was not implemented in this way, since its optimal feasible solution is easily derived from the assignment problem specification. The OPTIMAL program was used experimentally and differs in certain respects from the OPTIMAL function in Figure 5. For example, it uses a different procedure to obtain the initial feasible solution, and in the fathoming procedure, it uses rules in addition to that specified by Theorem 7 to hasten the fathoming of a partial solution.

In comparing the interference characteristics and profits of two data items, these rules identify conditions under which the assignment of one data item in an optimal feasible solution either cannot occur or can occur and is sufficient to ensure the assignment of the second data item.

A profit vector **p** and a data item interference matrix **C** are required to describe each assignment problem used in the comparison. One profit vector was constructed for use in all assignment problems; each element was effectively selected, with replacement, from a set of integers $\{i \mid 1 \leq i \leq 99\}$ with which there is associated a discrete uniform probability distribution. Four hundred **C** matrices were constructed for use in the comparison. Each **C** matrix is characterized by the matrix order, n, and the density, ρ, of nonzero matrix elements occuring off the main diagonal. The coordinates of each nonzero element were effectively selected, with replacement, from a set of integers $\{i \mid 1 \leq i \leq n\}$ with which there is associated a discrete uniform probability distribution. Five **C** matrices were constructed in this way for each (n, ρ) ordered pair, where $n \in \{16, 24\}$ and $\rho \in \{0.05, 0.15, \cdots, 0.95\}$. Ten **C** matrices were constructed for each (n, ρ) ordered pair, where $n \in \{32, 48, 64\}$ and $\rho \in \{0.05, 0.15, \cdots, 0.95\}$.

Figure 10 shows the OPTIMAL program average execution times required by OPTSOL to obtain optimal feasible solutions to the 400 assignment problems. The OPTIMAL program average execution time, t, depends not only on matrix order, but on matrix density and problem structure. When n is fixed, t varies greatly as a function of ρ; t usually decreases in value as ρ increases. For example, extreme values of t exhibited by OPTIMAL for $n = 64$, $m = 1$ are: $t = 358$ seconds at $\rho = 0.15$; and $t = 0.2$ seconds at $\rho = 0.95$.

When $m > 1$, the assignment problems passed by OPTSOL to OPTIMAL have the structure defined by Equations 19 and 21. The OPTIMAL program execution times for such problems are highly variable and considerably exceed the execution times required by unstructured problems having the same order.

Figures 11 and 12 show the ESTIMATE program average execution times required by ESTSOL2 and ESTSOL1 to obtain final feasible solutions to the 400 assignment problems. Since ESTSOL2 and ESTSOL1 generate the same input to ESTIMATE when $m = 1$, the curves in Figures 11 and 12 for $m = 1$ are identical. The average execution time for a single invocation of ESTIMATE depends on matrix order, but is reasonably insensitive to matrix density and problem structure. For example, extreme values of t exhibited by ESTIMATE for unstructured problems with $m = 1$, $n = 64$ are: $t = 118$ milliseconds at $\rho = 0.05$ and $t = 97$ milliseconds at $\rho = 0.55$. When $m = 2$, $n = 32$, the assignment problem input by ESTSOL1 to ESTIMATE has the structure defined by Equations 19

Figure 11 ESTIMATE average execution times required by ESTSOL2

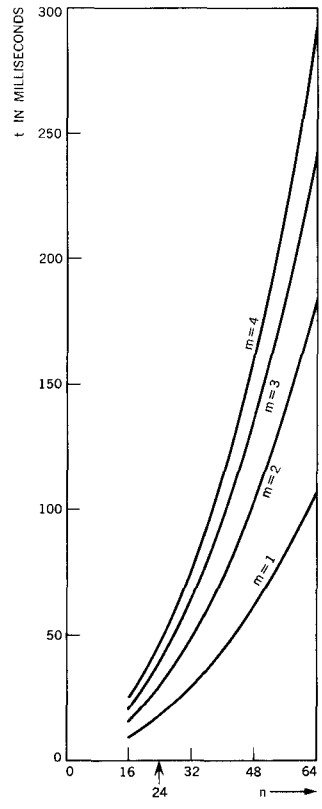


Figure 12 ESTIMATE average execution times required by ESTSOL1

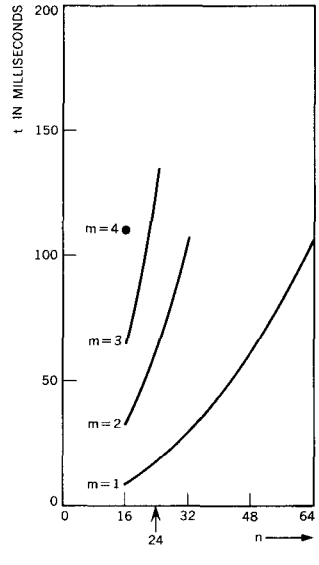
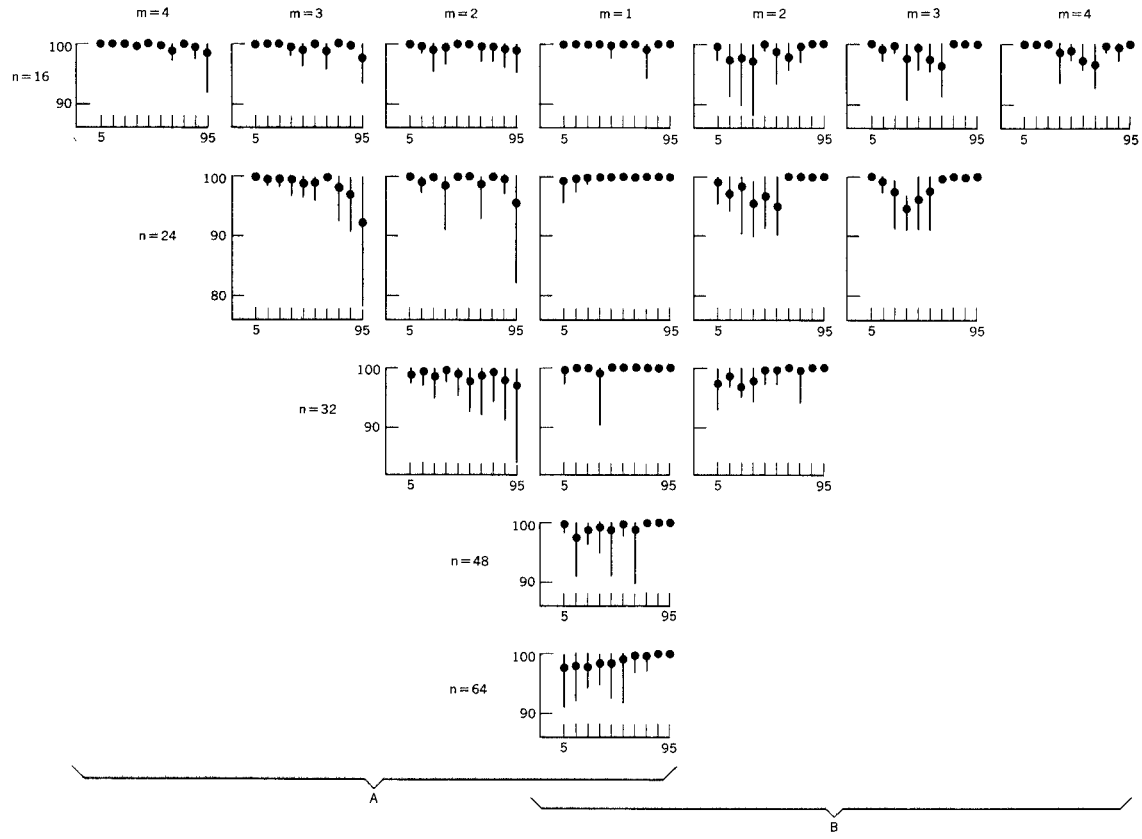


Figure 13 ESTSOL1 and ESTSOL2 profit comparisons



and 21 with a matrix order of 64. In this case, the extreme values of t exhibited by ESTIMATE are: $t = 123$ milliseconds at $\rho = 0.15$ and $t = 102$ milliseconds at $\rho = 0.85, 0.95$.

Curves closely approximating those in Figure 12 for $m > 1$ may be derived from the curve for $m = 1$. Let $t(n', m')$ denote the ESTIMATE average execution time required by ESTSOL1 to obtain feasible solutions to assignment problems with $n = n'$, $m = m'$. Using Equations 19 and 21, ESTSOL1 converts this problem into a new one, with a matrix order of $m'n'$, which is then input to ESTIMATE. It follows that: $t(n', m') \approx t(m'n', 1)$; this relationship may then be used to derive approximate curves for $m > 1$.

Figure 13A shows a comparison of the final feasible solution profits obtained by the ESTSOL1 and OPTSOL programs. The 400 assignment problems were partitioned into groups, each group corresponding to a distinct (n, ρ) ordered pair. For each problem in an (n, ρ) group, the ESTSOL1 final feasible solution profit was expressed as a percent of the OPTSOL optimal feasible solution profit. Figure 13A shows the high, low, and average percents obtained in each (n, ρ) group, for appropriate values of m. In the same

way, Figure 13B shows a comparison of the final feasible solution profits obtained by the ESTSOL2 and OPTSOL programs.

In these profit comparisons, ESTSOL1 and ESTSOL2 performance tends to degrade as either n or m increases. Nevertheless, the degradation is not severe for the type of assignment problems considered. The lowest percent obtained for a single assignment problem is: 78.5 percent by ESTSOL1 in the (24, 0.95) group when $m = 3$, and 88.3 percent by ESTSOL2 in the (16, 0.35) group when $m = 2$. The lowest average percent obtained for a group is: 92.5 percent by ESTSOL1 for the (24, 0.95) group when $m = 3$, and 94.5 percent by ESTSOL2 for the (24, 0.35) group when $m = 3$. In comparing ESTSOL1 with ESTSOL2, ESTSOL1 typically exhibits the superior profit performance when $\rho < 0.75$, while ESTSOL2 excels when $\rho \geq 0.75$.

Figure 14 shows a comparison of the final feasible solution profits obtained by the ONEONE, ESTSOL1 and OPTIMAL algorithms. The ESTSOL1 points are the (n, ρ) group average percents also shown in Figure 13A. The ONEONE points are (n, ρ) group average percents calculated using the ONEONE feasible solution profits. In addition, the ONEONE $(n, 0)$ group average percent is easily calculated from the profit vector and is included in Figure 14 as a reference point. The OPTSOL feasible solution profit usually decreases as ρ increases, thus bringing about the characteristic improvement observed in the ONEONE group average percent as ρ increases. In the limit, when $\rho = 1$, the final feasible solutions obtained by ESTSOL1, ONEONE, and OPTSOL have identical profits.

One might attribute the superior performance of ESTSOL1, with respect to ONEONE, to two factors:

1. The use of the **C** matrix by ESTIMATE to assign more than one data item in a feasible solution
2. The use of Equations 7 and 8 by ESTIMATE to determine the particular sequence in which data items should be assigned in a feasible solution

The RANDEST algorithm was implemented to determine the relative importance of these two factors. RANDEST differs from ESTSOL1 in the single respect that during ESTIMATE execution, a random permutation of integers is used to determine the particular sequence in which data items should be assigned in a feasible solution. For each of the 400 assignment problems, RANDEST was used to obtain eight final feasible solutions. The average profit of these eight solutions was expressed as a percent of the OPTSOL optimal feasible solution profit. Using these percents, the (n, ρ) group average percents were calculated; these are also shown in Figure 14. Factor 1 is dominant in assignment problems having low-density matrices and relatively large numbers of data items assigned in the final

solution. On the other hand, factor 2 is dominant in problems having high-density matrices; here, data item profit becomes important since relatively few data items are assigned in the final solution.

Extensions

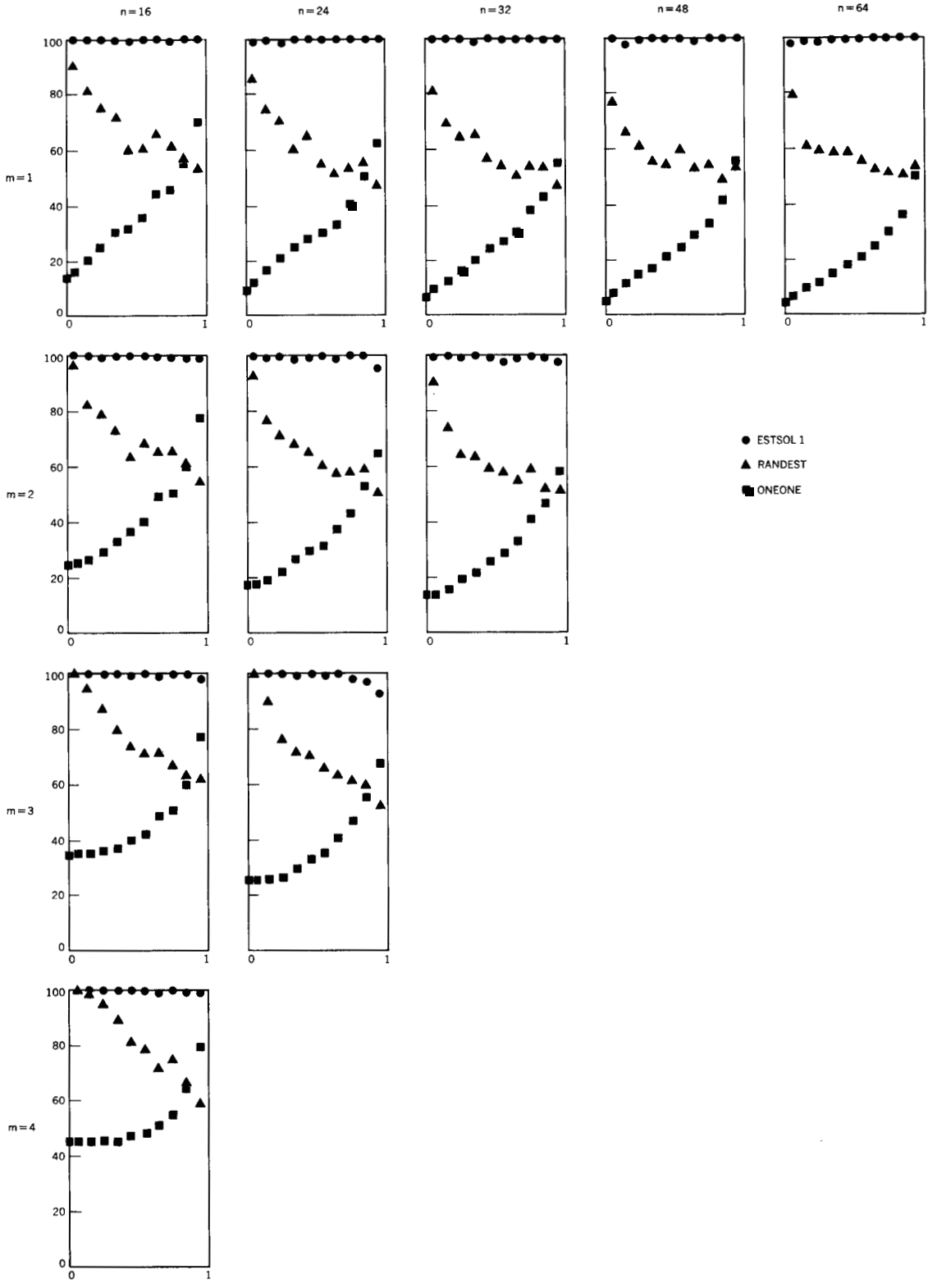
In previous sections, we discussed assignment methods based on the assumptions: that assignment may be considered a process involving local and global assignment; that global assignment may be considered a single-valued mapping; and that members of each class of registers subject to assignment have uniform characteristics. Here we indicate certain ways in which these assumptions might be weakened.

The global assignment methods described here effect a single-valued mapping of data items to registers in a region. These methods may also be used for local assignment to effect a single-valued mapping of data items to registers in a basic block. However, this approach does not recognize the considerable difference in the structural properties of the basic block and the region. It may be preferable to develop for local assignment a method using the distinctive structural properties of the basic block to effect in it a multiple-valued mapping of data items to registers.

Although global many-few assignment effects a single-valued mapping of data items to registers, it nevertheless may be used to effect a certain type of multiple-valued mapping. The concept motivating this extension is that in certain situations a given data item may be decomposed into a number of logically distinct data items. This set of nondecomposable data items may then replace the original data item as input to a global many-few assignment method. The resulting assignment is a single-valued mapping with respect to the nondecomposable data items, but it may be a multiple-valued mapping with respect to the original data item. Associated with each nondecomposable data item is a set of definitions of and references to the decomposable data item having the property that flow of control passes neither from a definition in the set to a reference not in the set, nor from a definition not in the set to a reference in the set. The identification of nondecomposable data items requires no information that is not already necessary in preparing input for global many-few assignment.

It seems natural to consider next whether a nondecomposable data item might itself be subdivided to provide still smaller input units for global assignment. If the data item were subdivided into n parts, the resulting $(2^n - 1)$ distinct partial assignments might then replace it as input to a global many-few assignment method. The corresponding interference matrix must be initialized to ensure that each of the n parts is assigned, at most, once to, at most, one

Figure 14 ESTSOL1, ONEONE, and RANDEST profit comparisons



register. With this approach, the challenge is to develop a method of subdividing nondecomposable data items that will keep the total number of partial assignments within reasonable bounds, and yet will make possible a reasonable improvement in the profit of the resulting assignment.

It may happen that members of a class of registers do not have uniform characteristics in a region of the program subject to global many-few assignment. Particular registers may be required during the execution of certain instructions. Requirements imposed by the programmer, the operating system, or the compiler itself may restrict the availability of particular registers to specific parts of the region. If the use of a register is so restricted, an artificial data item corresponding to the register may be added to the problem to permit assignment of data items to the register while its use is unrestricted. This artificial data item interferes with all other artificial data items and with data items that are active while the use of the register is restricted. Its profit is set high enough to ensure its selection in any final feasible solution to the global many-few assignment problem.

It may also happen that adjacent registers are coupled together during the execution of certain instructions. In this case, it may be desirable to perform global assignment in two steps. First, use the global many-few assignment method to map data items into the coupled registers. Then, use the technique described in the previous paragraph with the global many-few assignment method to map remaining data items into available parts of the uncoupled registers.

Global many-few assignment may also be a useful point of departure in studying the assignment of data items to main storage. Let the entire program be the region of interest. Consider for assignment data items having the same length, and associate with each data item a unit profit. Then global many-one assignment effects a mapping of many data items into a single main storage area. However, repeated assignments of this type do not necessarily minimize the length of the total main storage area into which the data items are mapped. The general problem, in which data items have varying lengths, is to find an assignment of data items to a main storage area of minimum length. It seems likely that efficient assignment methods, using the interference matrix and the data item lengths, can be developed to obtain close approximations to the optimal solution to this problem.

Summary

In this paper, we described three methods of globally assigning data items to registers: one-one, many-one, and many-few. Each method is a particular single-valued mapping of a set of data

items into a set of registers. We developed a formulation of each method as an integer programming problem, and showed the one-one and many-one global assignment problems to be special cases of the global many-few assignment problem.

Next we described three algorithms for obtaining feasible solutions to the many-one and many-few global assignment problems. The OPTSOL algorithm uses a branch-and-bound procedure to obtain optimal feasible solutions to these problems, and we proved that these branch-and-bound solutions are indeed optimal. The ESTSOL1 and ESTSOL2 algorithms obtain possibly nonoptimal feasible solutions to these problems.

The ESTSOL1 and ESTSOL2 algorithms have identical execution times for global many-one assignment problems; however, ESTSOL2 has the shorter execution time for assignment problems with multiple registers. In the range of problems analyzed, both alogrithms seem fast enough to be considered for inclusion in an optimizing compiler. Also, the solution profits of both algorithms are almost always within ten percent of the optimal profit and are significantly better than that of the corresponding global one-one assignment problem.

Certain extensions in the use of the global many-few assignment method may increase the profitability of the final assignments.

Appendix A: Proof of Theorem 2

If $\mathcal{C}(\sum^f) = 1$, then $\sum^f$ satisfies the definition of sequence nonredundancy since each complete solution fathomed in $\sum^f$ clearly occurs as the completion of exactly one fathomed partial solution. This completes the proof for the case when $\mathcal{C}(\sum^f) = 1$.

Now assume that $\mathcal{C}(\sum^f) > 1$. The proof is by mathematical induction. First we show that $\{S_1^p\}$ is nonredundant. Then, assuming that $\{S_1^p, \cdots, S_i^p\}$ is nonredundant, we show that $\{S_1^p, \cdots, S_{i+1}^p\}$ must be nonredundant. The Principle of Mathematical Induction then assures us that $\sum^f$ itself must be nonredundant.

The proof that $\{S_1^p\}$ is nonredundant is identical to that used for the case when $\mathcal{C}(\sum^f) = 1$.

Now assume that the sequence $\{S_1^p, \cdots, S_i^p\}$ is nonredundant. S_{i+1}^p is constructed from S_i^p by application of step 5 in Table 1, optionally followed by (repetitive) application of step 4. The structure of these consecutive fathomed partial solutions may be represented as the four cases shown in Table 3.

In Cases 1 and 2, S_{i+1}^p is nonredundant with respect to S_i^p because $s_j \in S_{i+1}^p$ is the complement of $s_j' \in S_i^p$. S_i^p is itself nonredundant

Table 3 Structure of consecutive fathomed partial solutions

In every case:

$$s_m = s_m' \quad \text{for} \quad 1 \leq m < j$$

$$s_j' > 0$$

$$s_j = -s_j'$$

Case 1: $S_i^p = \{s_1', \cdots, s_j'\}$

$S_{i+1}^p = \{s_1, \cdots, s_j\}$

Case 2: $S_i^p = \{s_1', \cdots, s_j'\}$

$S_{i+1}^p = \{s_1, \cdots, s_j, \cdots, s_k\}$

where $s_m > 0 \quad \text{for} \quad j < m \leq k$

Case 3: $S_i^p = \{s_1', \cdots, s_j', \cdots, s_k'\}$

$S_{i+1}^p = \{s_1, \cdots, s_j\}$

where $s_m' < 0 \quad \text{for} \quad j < m \leq k$

Case 4: $S_i^p = \{s_1', \cdots, s_j', \cdots, s_k'\}$

$S_{i+1}^p = \{s_1, \cdots, s_j, \cdots, s_l\}$

where $s_m' < 0 \quad \text{for} \quad j < m \leq k$

$s_m > 0 \quad \text{for} \quad j < m \leq l$

with respect to all previous S_n^p, $1 \leq n < i$, because of the presence of complemented elements in S_i^p in the element sequence $\{s_1', \cdots, s_{j-1}'\}$. Since this element sequence also appears in S_{i+1}^p, S_{i+1}^p is nonredundant with respect to all previous S_n^p, $1 \leq n \leq i$. Since, in addition, $\{S_1^p, \cdots, S_i^p\}$ is a nonredundant sequence, it follows that $\{S_1^p, \cdots, S_{i+1}^p\}$ is a nonredundant sequence in Cases 1 and 2.

In Cases 3 and 4, S_{i+1}^p is nonredundant with respect to S_i^p because $s_j \in S_{i+1}^p$ is the complement of $s_j' \in S_i^p$. Since s_j' first occurs in S_g^p, S_{i+1}^p is also nonredundant with respect to all previous S_m^p, $g \leq m < i$. S_g^p is itself nonredundant with respect to all previous S_n^p, $1 \leq n < g$, because of the presence of complemented elements in S_g^p in the element sequence $\{s_1', \cdots, s_{j-1}'\}$. Since this element sequence also appears in S_{i+1}^p, S_{i+1}^p is nonredundant with respect to all previous S_n^p, $1 \leq n \leq i$. Since, in addition, $\{S_1^p, \cdots, S_i^p\}$ is a nonredundant sequence, it follows that $\{S_1^p, \cdots, S_{i+1}^p\}$ is a nonredundant sequence in Cases 3 and 4. This completes the proof.

Appendix B: Proof of Theorem 3

Step 5 of Table 1 is entered only when a partial solution has been fathomed, and it is the only step that creates the negative elements found in the $S_i^p \in \sum^f$. The negative elements in S_i^p, suitably in-

terpreted, provide a record of the complete solutions implictly enumerated by the fathoming process in previous steps. The interpretation is this. If:

$$S_i^p = \{s_1, \cdots, s_j, \cdots, s_k\}$$

and $s_j < 0$, then the $2^{(n-i)}$ completions of

$$S^p = \{s_1, \cdots, -s_j\}$$

have been implicitly enumerated in the fathoming of S_m^p, $1 \leq m < i$. In this context, to prove the theorem it is sufficient to prove that:

1. Every $S_i^p \in \sum^f$ exhibits a complete record of the complete solutions implicitly enumerated in the fathoming of all previous S_j^p, $1 \leq j < i$;
2. The OPTIMAL algorithm termination criterion is satisfied only when the record indicates that all 2^n complete solutions have been implicitly enumerated.

The proof of part 1 is by mathematical induction. First we show that S_1^p exhibits a valid record. Then, assuming that S_i^p exhibits a valid record, we show that S_{i+1}^p must exhibit a valid record. The Principle of Mathematical Induction then assures us that every $S_i^p \in \sum^f$ must exhibit a valid record.

Step 5 is entered for the first time when S_1^p is fathomed. S_1^p contains no negative elements because such elements are created only by step 5, and this step has not been entered previously. The absence of negative elements implies that no complete solutions have been previously enumerated. This is a valid record for S_1^p, since it is the first fathomed partial solution.

Now assume that S_i^p exhibits a valid record. S_{i+1}^p is constructed from S_i^p by application of step 5, optionally followed by (repetitive) application of step 4. The structure of these consecutive fathomed partial solutions may be represented as the four cases shown in Table 3.

In Cases 1 and 2, all negative elements in S_i^p appear in S_{i+1}^p by construction, so that the valid record of S_i^p is propagated to S_{i+1}^p. It remains only to record in S_{i+1}^p the fact that S_i^p itself has been fathomed; step 5 does this by making $s_j \in S_{i+1}^p$ the complement of $s_j' \in S_i^p$. It follows that S_{i+1}^p exhibits a valid record in Cases 1 and 2.

In Cases 3 and 4, all $s_m' \in S_i^p$, $j < m \leq k$ are negative. S_i^p has been fathomed so that all completions of

$$\{s_1', \cdots, s_j', \cdots, s_k'\} \tag{22}$$

have been implicitly enumerated. In addition, $s_k' < 0$ indicates that all completions of

$$\{s_1', \cdots, s_i', \cdots, -s_k'\} \tag{23}$$

have been previously implicitly enumerated. Equations 22 and 23 comprise all completions of

$$\{s_1', \cdots, s_i', \cdots, s_{k-1}'\} \tag{24}$$

so that, with no loss of information, we may delete s_k' from S_i^p. This argument applies in turn to each right-hand $s_m' \in S_i^p$, $j < m \leq (k-1)$ so that S_i^p assumes the form:

$$S_i^p = \{s_1', \cdots, s_i'\}$$

Cases 3 and 4 now have the form of Cases 1 and 2, to which the previous arguments apply. This completes the proof of part 1.

The OPTIMAL algorithm terminates when a partial solution S_i^p has been fathomed that has either of the forms:

$$S_i^p = \{s_1, \cdots, s_j\}$$

where $s_k < 0$ for $1 \leq k \leq j$, or:

$$S_i^p = \varnothing$$

In the former case, the argument of Equations 22 through 24 is applied in turn to delete each right-hand negative element, so that S_i^p assumes the form of the latter case. Since a fathomed partial solution containing no elements has 2^n completions, we conclude that in either case all 2^n complete solutions have been implicitly enumerated when termination occurs. This completes the proof of part 2.

Appendix C: Proof of Theorem 8

Let F^* be a partition of F^f:

$$F^* = \{\cdots, F_i, \cdots\}$$

where:

$$F_i = \{f \mid f \in F^f, g \in F_i, c_{fg} = 1 \quad \text{for} \quad g \neq f\}$$

It is possible to construct the partition F^* with the following algorithm. For $f_1 \in F^f$, we define $F_1 = \{f_1\}$. The remaining $f_i \in F^f$, $i > 1$, are to be assigned to $F_j \in F^*$ in the index sequence: $\{2, 3, \cdots, m\}$. Each f_i, $2 \leq i \leq m$, is assigned to an existing nonempty F_j, $1 \leq j \leq t$, if it interferes with all $f \in F_j$; otherwise we define a new subset, $F_{t+1} = \{f_i\}$.

D is an upper bound on the cardinality of F^*. The proof follows. Any given f_i in the sequence: $\{f_1, \cdots, f_m\}$ does *not* interfere with

$$m_i = (i-1) - \sum_{j=1}^{i} c_{ij}^*$$

of the $(i - 1)$ data items already assigned to sets $F_j \in F^*$. In the worst case, these m_i data items would be assigned to m_i distinct sets; in this case,

$$m_i + 1 = i - \sum_{j=1}^{i} c_{ij}^*$$

sets would be sufficient to accomplish the assignment of f_i to some $F_j \in F^*$. It follows that for the partitioning algorithm described,

$$D = \max_{i=1,\cdots,m} \left\{ i - \sum_{j=1}^{i} c_{ij}^* \right\}$$

sets would be sufficient for the complete partitioning of F^f into F^*. Thus

$$D = \max_{i=1,\cdots,m} \left\{ i - \sum_{j=1}^{i} c_{ij}^* \right\} \geq \mathcal{C}(F^*) \tag{25}$$

Now let S^c determine a feasible completion of S^p, and suppose that $f \in F_i$ is assigned in S^c. No $g \in F_i$, $g \neq f$, may be assigned in S^c because of the existence of total interference among the data items in F_i. On the other hand, there may exist as many as $\mathcal{C}(F^*)$ data items, each in a distinct F_i that may be assigned in S^c. From this and the definition of T, it follows that:

$$\mathcal{C}(F^*) \geq \mathcal{C}(T)$$

Substituting this result in Equation 25 yields Equation 12, which we set out to prove.

Appendix D: Proof of Theorem 9

Let F^* be a partition of F^f:

$$F^* = \{\cdots, F_i, \cdots\}$$

where:

$$F_i = \{f \mid f \in F^f, g \in F_i, c_{fg} = 0\}$$

It is possible to construct the partition F^* with the following algorithm. For $f_m \in F^f$, we define $F_1 = \{f_m\}$. The remaining $f_i \in F^f$, $i < m$, are to be assigned to $F_j \in F^*$ in the index sequence: $\{m - 1, m - 2, \cdots, 1\}$. Each f_i, $1 \leq i < m$, is assigned to an existing nonempty F_j, $1 \leq j \leq t$, if it does not interfere with any $f \in F_j$; otherwise we define a new subset, $F_{t+1} = \{f_i\}$.

R is an upper bound on the cardinality of F^*. The proof follows. Any given f_i in the sequence: $\{f_m, \cdots, f_1\}$ interferes with

$$m_i = \sum_{j=i}^{m} c_{ij}^*$$

of the $(m - i)$ data items already assigned to sets $F_j \in F^*$. In the worst case, these m_i data items would be assigned to m_i distinct sets; in this case,

$$m_i + 1 = 1 + \sum_{j=i}^{m} c_{ij}^*$$

sets would be sufficient to accomplish the assignment of f_i to some $F_j \in F^*$. It follows that for the partitioning algorithm described,

$$R = 1 + \max_{i=1,\cdots,m} \left\{ \sum_{j=i}^{m} c_{ij}^* \right\}$$

sets would be sufficient for the complete partitioning of F^f into F^*. Thus

$$R = 1 + \max_{i=1,\cdots,m} \left\{ \sum_{j=i}^{m} c_{ij}^* \right\} \geq \mathcal{C}(F^*) \tag{26}$$

Now let T' be a set of elements $S_i^c \in T'$ where:

$$\begin{aligned} S_i^c = \{ s \mid\ & |s| \in F, \\ & (s \in F_i) \Rightarrow (s > 0), \\ & (|s| \notin F_i) \Rightarrow (s < 0) \} \end{aligned} \tag{27}$$

and:

$$\mathcal{C}(T') = \mathcal{C}(F^*) \tag{28}$$

Each $f \in F^f$ is assigned in exactly one $S_i^c \in T'$; this follows from Equation 27 and the fact that F^* is a partition of F^f. In addition, each $S_i^c \in T'$ determines a feasible completion of S^p since $F_i \subseteq F^f$ and, by construction, there exists no interference among the data items in F_i. On the other hand, since there is no reason to suppose that the cardinality of T' is a minimum, it follows that:

$$\mathcal{C}(T') \geq \mathcal{C}(T)$$

From this, Equation 28, and Equation 26 we obtain Equation 13, which we set out to prove.

CITED REFERENCES AND FOOTNOTES

1. A path is a sequence of arcs of a graph such that the terminal vertex of each arc coincides with the initial vertex of the succeeding arc.
2. $\mathcal{C}(N)$ represents the cardinality of the set N.
3. S. Rosen, R. A. Spurgeon, and J. K. Donnelly, "PUFFT—The Purdue University fast FORTRAN translator," *Communications of the ACM* **8,** No. 11, 661–666 (November 1965).
4. P. W. Shantz, R. A. German, J. G. Mitchell, R. S. K. Shirley, and C. R. Zarnke, "WATFOR—The University of Waterloo FORTRAN IV compiler," *Communications of the ACM* **10,** No. 1, 41–44 (January 1967).
5. F. E. Allen, "Program optimization," M. I. Halpern and C. J. Shaw (editors), *Annual Review in Automatic Programming* **5,** 239–307, Pergamon Press, New York, New York (1969).
6. E. S. Lowry and C. W. Medlock, "Object code optimization," *Communications of the ACM* **12,** No. 1, 13–22 (January 1969).
7. J. Cocke and J. T. Schwartz, *Programming Languages and Their Com-*

pilers, Courant Institute of Mathematical Sciences, New York University, New York, New York (1969).
8. L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd, "Index register allocation," *Journal of the Association for Computing Machinery* **13,** No. 1, 43–61 (January 1966).
9. M. L. Balinski, "Integer programming: methods, uses, computation," *Management Science* **12,** No. 3, 253–313 (November 1965).
10. A. M. Geoffrion, "Integer programming by implicit enumeration and Balas' method," *SIAM Review* **9,** No. 2, 178–190 (April 1967).
11. L. G. Mitten, "Branch-and-bound methods: general formulation and properties," *Operations Research* **8,** No. 1, 24–34 (January-February 1970).
12. M. L. Balinski and K. Spielberg, "Methods for integer programming: algebraic, combinatorial, and enumerative," *Progress in Operations Research: Relationship Between Operations Research and the Computer* **3,** 197–292, J. S. Aronofsky (editor), John Wiley and Sons, Inc., New York, New York (1969).