A MULTIPLIER-ADJUSTMENT-BASED BRANCH-AND-BOUND ALGORITHM FOR THE PARTITIONING PROBLEM

Thomas Justin Chan
James C. Bean
Candace Arai Yano

October 1987

Technical Report 87-35

A MULTIPLIER-ADJUSTMENT-BASED BRANCH-AND-BOUND ALGORITHM FOR THE SET PARTITIONING PROBLEM

Thomas Justin Chan[†]

James C. Bean[‡]

Candace Arai Yano[‡]

October 1987

[†] Department of Operations Research and Engineering Management, School of Engineering and Applied Sciences, Southern Methodist University, Dallas, TX 75275

[‡] Department of Industrial and Operations Engineering, The University of Michigan, Ann Arbor, Michigan 48109-2117

Subject Classification: 628 Set Partitioning Problem, 630 multiplier adjustment method bounding procedure with variable elimination.

Abstract

We introduce a branch-and-bound algorithm for solving the set partitioning problem (SPP). The new algorithm employs a multiplier adjustment method bounding procedure, and a new branching strategy which results in relatively small search trees. Typically, only three subproblems are created from each branching. We also extend the results of variable elimination to the SPP. Results for ten moderate-sized problems show that, on average, 94.4% of the variables can be eliminated without loss of optimality. Elimination of the variables reduces the computational effort of the bounding procedure and increases the likelihood of fathoming the subproblems. On average, the computation times of this algorithm are 9.5 times faster than known techniques on problems tested. Moreover, this ratio is observed to increase as the size of the problem increases.

A Multiplier-Adjustment-Based Branch-and-Bound Algorithm for the Set Partitioning Problem

The set partitioning problem (SPP) is a zero-one integer program formulated as follows:

$$
\begin{array}{llll}
\text{(P)} & \text{minimize} & \sum_{j=1}^{n} C_j x_j & \\
& \text{s.t.} & \sum_{j=1}^{n} a_{ij} x_j = 1 & i \in I \qquad (1)\\
& & x_j = \{0,1\} & j \in J \\
& \text{where} & a_{ij} \in \{0,1\} & i \in I,\ j \in J \\
& & I = \{1, 2, \ldots, m\}, \quad J = \{1, 2, \ldots, n\} &
\end{array}
$$

The SPP has been the focus of study by many researchers because of its simple structure and numerous practical applications. Among the applications described in the literature are: crew scheduling (Marsten and Shepardson 1981), truck scheduling (Balinski and Quant 1964), information retrieval (Day 1965), circuit design (Root 1964), capacity balancing (Steinman and Schwinn 1969), capital investment (Valenta 1969), facility location (Revelle, Marks and Liebman 1970), political districting (Garfinkel and Nemhauser 1970) and radio communication planning (Thuve 1981). Other applications of the SPP are given in the surveys by Garfinkel and Nemhauser (1972), and Salkin (1975).

The two best published approaches for the SPP are implicit enumeration and simplex-based cutting plane. (A survey of these and other approaches is provided by Balas and Padberg 1979.) Implicit enumeration is the more promising and more widely used of the two because it takes full advantage of the special structure of the SPP. Branching is usually performed by either *fixing* a variable $x_j = 1$ to satisfy a particular constraint (Pierce 1968; and Garfinkel and Nemhauser 1969) or restricting the set from which a variable may be selected to satisfy a particular constraint (Marsten 1974).

One means of enhancing the performance of implicit enumeration is to reduce the size of the search tree. This is usually achieved by calculating a lower bound on the cost of completion for each partial solution. To accomplish this, Michaud (1972) solves a LP in the free variables for each partial solution. To avoid solving a LP for each partial solution, Pierce and Lasky (1973) and Lu (1970) solve a knapsack problem obtained by adding up all the unsatisfied constraints, while Christofides and Korman (1973) solve an auxiliary problem using dynamic programming techniques.

A more successful bounding technique was developed by Etcheberry (1977), which uses Lagrangian relaxation (Fisher 1981 and Geoffrion 1974) and subgradient optimization (Sandi 1979). Marsten and Shepardson (1981) combine Marsten's branching strategy with Etcheberry's bounding strategy in an airline crew scheduling application. This hybrid algorithm is found to perform better than Marsten's original algorithm (Gerbracht 1978). Currently, SETPAR (Marsten, Muller and Killion 1979) is considered to be the best published algorithm for solving the SPP (Fisher and Kedia 1986).

This paper introduces a more efficient algorithm for solving the set partitioning problem. The algorithm (BB-SPP) employs a multiplier adjustment based bounding method (called MAM) and a new branching strategy, which we developed to complement MAM. See Chan and Yano (1987) for details of MAM.

Before describing the new algorithm, we define the notation and provide an overview in section 1. We present the bounding strategy in section 2, and give the motivation for and details of the branching strategy in section 3. In section 4, we describe the branch-and-bound algorithm. In section 5, we explain how the concept of variable elimination can be extended to the SPP and suggest how it can be applied. In section 6, we give the computational results for the new algorithm and describe how we implemented several improvement strategies and variable elimination. For comparison, the computational results for SETPAR are also given.

1. Notation and Overview

Before stating the algorithm, we need to define the following notation:

$F = \{ j \in J : x_j \text{ is fixed to be } 1 \}$

$\bar{I} = \{ i \in I : \text{constraint } i \text{ is free} \}$

$\bar{J} = \{ j \in J : \text{variable } j \text{ is free} \}$

$I_j = \{ i \in \bar{I} : a_{ij} = 1 \}$

$J_i = \{ j \in \bar{J} : a_{ij} = 1 \}$

LIST = list of candidate problems to be investigated

$U = (u_1, u_2, \ldots, u_m)$ = dual solution vector

LB = lower bound of current subproblem

$SOL = \{ j \in \bar{J} : x_j = 1 \}$

UB = upper bound (objective value of incumbent)

$n_i = | SOL \cap J_i |$

$S_0 = \{ i \in \bar{I} : n_i = 0 \}$

$S_1 = \{ i \in \bar{I} : n_i = 1 \}$

$S_2 = \{ i \in \bar{I} : n_i \geq 2 \}$

The branch-and-bound algorithm consists of two main components: branching and bounding. The branching component creates new subproblems by *fixing* certain variables to equal one or zero. When a variable is fixed to equal one, certain variables and constraints are deleted. Hence, each new subproblem contains fewer free variables and constraints than the subproblem from which in branches. The remaining variables and constraints are referred to as the *free* variables and constraints, respectively.

The index set F represents the variables which are fixed to equal one for the current subproblem. Sets $\bar{I}$ and $\bar{J}$ contain the indexes of the free constraints and variables, respectively, for the current subproblem. Subproblems corresponding to the unfathomed terminal nodes of the search tree are referred to as candidate problems. Subproblems created during branching are referred to as candidate subproblems. If a candidate subproblem is not fathomed, then it is stored in LIST and may be investigated during future branching.

The bounding component attempts to obtain a tight lower bound to the candidate subproblems. This is accomplished by applying an iterative algorithm, MAM, to each subproblem. At each iteration of MAM, a feasible dual solution (U) is obtained for the LP relaxation of the subproblem. This dual solution is then used to determine a corresponding lower bound (LB) and an integer solution. The integer solution is represented by the set SOL, which contains the indexes of the variables equalling one. If the integer solution is feasible to the subproblem, then SOL$\cup$F represents a feasible solution to the original problem P. The current best feasible solution of P is referred to as the incumbent. The objective value of the incumbent provides the current upper bound, UB.

If $i \in I_j$ and $j \in$ SOL, then we say constraint i is *covered* by variable j. The number of variables covering constraint i is indicated by n_i. Given an integer solution to a subproblem, the free constraints can be classified into three groups: 1) those which are *under-covered* (i.e., not covered by any variable); 2) those which are *tight* (i.e., covered by exactly one variable); and 3) those which are *over-covered* (i.e., covered by two or more variables). These three groups of constraints are represented by the index sets S_0, S_1 and S_2, respectively.

For notational simplicity, we also assume the constraints are indexed such that $|J_{i_1}| < |J_{i_2}|$ for any $i_1, i_2 \in I : i_1 < i_2$ at the beginning of the algorithm, where $\bar{J} = J$.

2. Bounding Strategy

Our bounding strategy is based on MAM, which overcomes many of the weaknesses inherent in subgradient optimization. For example, MAM requires no stepsizes nor an upper bound, and the lower bound obtained at each iteration is monotonically nondecreasing. When applied to a subproblem, MAM attempts to solve the dual of the LP relaxation as shown below:

$$\begin{array}{lll} \textbf{maximize} & \sum_{i \in \bar{I}} u_i & \\ \text{s.t.} & \sum_{i \in \bar{I}} a_{ij} u_i \le C_j & j \in \bar{J} \\ & u_i \ \textbf{unrestricted} & i \in \bar{I} \end{array} \qquad (2)$$

At each iteration, MAM obtains a feasible dual solution U and a corresponding lower bound LB, which equals $\sum_{i \in \bar{I}} u_i + \sum_{j \in F} C_j$. Based on U, a primal integer solution is also obtained by setting the integer variables corresponding to tight dual constraints in (2) to equal one. If this integer solution is feasible, MAM terminates and the subproblem is fathomed. If its objective value is less than UB, then the incumbent is also updated. On the other hand, if the integer solution is infeasible, then LB is examined. If LB is greater than UB, then the subproblem is fathomed; otherwise, it is stored in LIST.

A formal statement of the MAM algorithm and a brief description indicating how MAM was incorporated into our branch-and-bound algorithm are given in the appendix.

3. Branching Strategy

The branching strategy was developed to complement the bounding method MAM. We have observed that when MAM is applied to problem P, SOL contains many variables which equal one in the optimal solution, even after only one iteration. In particular, the variables in the set $\{ j \in \mathrm{SOL} : I_j \subseteq S_1 \}$ usually equal one in the optimal solution. Moreover, for each $i \in S_2$, $\bar{j}_i$ (the variable which covers constraint i in the optimal solution) is usually contained in the set $SOL_i = SOL \cap J_i$. If $\bar{j}_i$ is not in SOL_i, then it usually becomes an element of SOL when MAM is applied to the subproblem with SOL_i deleted from $\bar{J}$.

To take advantage of this characteristic of SOL, the algorithm branches by fixing selected variables in $\{ j \in SOL : I_j \cap S_2 \neq \emptyset \}$ to equal one or zero. To determine which variables to fix, the algorithm first identifies the constraint having the smallest value of $| J_i |$ among the least over-covered constraints. This constraint is indexed by p. Then the variables in SOL covering constraint p are the set of candidates for branching. The reasons for this selection rule are: 1) Choosing a least over-covered constraint minimizes the number of branches created, especially at the top of the search tree, which in turn may reduce the computational effort of the algorithm; and 2) The smaller the value of $| J_i |$, the more difficult it is to cover the corresponding constraint.

Next, the algorithm creates $q+1$ candidate subproblems, where $q = |SOL_p|$. Initially, the values of $\bar{I}$, $\bar{J}$, F, SOL and U for each candidate subproblem take on the corresponding values of the parent candidate problem. The first q candidate subproblems are created by fixing a unique variable in SOL_p to equal one. Fixing a variable $x_j = 1$ reduces a candidate subproblem by $| I_j |$ constraints and $| \bigcup_{i \in I_j} J_i |$ variables.

Candidate subproblem $q+1$ is created by fixing all variables in SOL_p to equal zero. This is necessary in case $\bar{j}_p$ is not in SOL_p. The formal statements of the branching component are given below:

```
Step B1. (a) Let $p = \min_{i \in S_2} \{ i : n_i = \underline{n} \}$ where $\underline{n} = \min_{i \in S_2} \{ n_i \}$.
         (b) Let $SOL_p = \{ j_1, j_2, j_3, \ldots, j_q \}$ where $q = | SOL_p |$.

Step B2. For candidate subproblem $k \in \{1,2,3,\ldots,q\}$:

              (a) Set $\bar{I}$, $\bar{J}$, F, SOL and U to the corresponding values of
                  the parent candidate problem.
```

```
    (b) Set $F = F \cup \{j_k\}$; $\bar{I} = \bar{I}\backslash I_{j_k}$ and $\bar{J} = \bar{J}\backslash(\bigcup_{i \in I_{j_k}} J_i)$.
For candidate subproblem q+1:
    (a) Set $\bar{I}$, $\bar{J}$, F, SOL and U to the corresponding values of
        the parent candidate problem.
    (b) Set $\bar{J} = \bar{J}\backslash SOL_p$.
```

This branching strategy eliminates the need to branch into $| J_p |$ candidate subproblems each time. Since each new candidate subproblem requires the application of MAM to obtain a lower bound, the strategy reduces the computational effort tremendously, especially when $| J_p |$ is large. In the worst case, the strategy is equivalent to branching into $| J_p |$ candidate subproblems for each candidate problem investigated. However, for all the problems that we have encountered so far, the resultant search trees are small.

4. The Branch-and-bound Algorithm

The new algorithm consists of the following five steps:

```
Step 1. (a) Set $\bar{I} = I$, $\bar{J} = J$, $F = \emptyset$, LB = 0, $U = \bar{0}$ and UB = $\infty$.
        (b) Apply MAM.
            If SOL is optimal, terminate.
            Otherwise, go to step 3.
Step 2. (a) If LIST is empty, terminate.
            Otherwise, retrieve candidate problem with smallest LB.
        (b) If LB >= UB, terminate.
Step 3. Determine branching parameters.    (See step B1 in section 3.)
Step 4. Create new candidate subproblems.  (See step B2 in section 3.)
Step 5. (a) For each candidate subproblem:
            (i) Check feasibility.
        (ii) Apply MAM.   (See the appendix.)
        (iii) If subproblem is not fathomed during (i) or (ii),
              then store it in LIST.
    (b) Go to step 2.
```

In step 1, the variables are initialized and then MAM is applied to problem P. If the integer solution obtained by MAM is optimal, then the algorithm terminates. Otherwise, the algorithm goes to step 3 to begin the branching process.

In step 2, the candidate problem with the smallest LB is retrieved from LIST. If LIST is empty or LB of the candidate problem retrieved is greater than or equal to UB, then the algorithm terminates and the current incumbent is optimal to P. However, if there is no incumbent, then P is infeasible.

Steps 3 and 4 constitute the branching component of the algorithm. Step 3 determines the branching parameters, while step 4 uses these parameters to determine how the new candidate subproblems are to be created. (See section 3 for details.)

In step 5, each candidate subproblem is first checked for feasibility. This is necessary because certain variables are deleted when a candiate subproblem is created at step 4. (The details of the feasibility test are given shortly.) If the subproblem is found to be infeasible, then it is fathomed. Otherwise, MAM is applied to it. If the subproblem is not fathomed within the MAM algorithm, then it is stored in LIST. After all the candidate subproblems are examined, the algorithm returns to step 2 and repeats step 2 to 5 until one of the termination criteria in step 2 is satisfied.

Feasibility Test

The following feasibility test is applied to each new candidate subproblem at step (5ai) of the new algorithm:

```
For each $i \in \bar{I}$ in turn:
  Step F1. If $| J_i | = 0$, fathom.
Step F2. If $|J_i| = 1$:      (Suppose $J_i = \{ k \}$.)

          (a) Set $F = F \cup \{k\}$ and $\bar{I} = \bar{I} \setminus I_k$.

          (b) If $\bar{I} = \emptyset$:
              (i) If $LB = \sum_{j \in F} C_j < UB$,
                    set UB = LB and update incumbent.

              (ii) Fathom subproblem.

          (c) $\bar{J} = \bar{J} \setminus ( \bigcup_{i \in I_k} J_i )$.

          (d) Repeat feasibility test.
```

For any free constraint i, if J_i is empty, then the subproblem is infeasible and can be deleted without further consideration. If J_i contains only one variable, then the variable is fixed to equal one and $\bar{I}$ is reduced accordingly at step (F2a). After $\bar{I}$ is reduced, if all free constraints are satisfied (i.e., $\bar{I} = \emptyset$), then F becomes a feasible solution to P. If the corresponding objective value is lower than UB, then UB is updated and F becomes the new incumbent. Next, the subproblem is fathomed.

If any free constraints remain at step (F2b), then $\bar{J}$ is reduced accordingly at step (F2c). However, after $\bar{J}$ is reduced, $|J_i|$ may become zero for an free constraint i that has already been examined. Hence, the feasibility test must be repeated from the beginning.

5. Variable Elimination

Variable elimination for the Multiple Choice Integer Program was presented formally by Sweeney and Murphy (1979), and then implemented within a branch-and-bound framework for the multi-item scheduling problem (Sweeney and Murphy 1981). We extend this idea to the SPP. The justification is provided by Theorem 1 below. For the SPP, we define the reduced cost for variable j as $C_j - \sum_{i=1}^{m} a_{ij}u_i$. Hereafter, the optimal multiplier vector is denoted as $\tilde{U}$; the optimal reduced cost for variable j as $\tilde{C}_j$; the LP relaxation of P as $\overline{P}$; the dual of $\overline{P}$ as D; the feasible region for problem $(\cdot)$ as $F(\cdot)$; and the complement of $F(\cdot)$ as $\overline{F}(\cdot)$.

Theorem 1 : If $\tilde{C}_j > v(P) - v(\overline{P})$, then $x_j = 0$ in any optimal solution to P.

Proof : Suppose $\tilde{X}$ is an optimal solution to P with $\tilde{x}_j = 1$ and $\tilde{C}_j > v(P) - v(\overline{P})$, then using vector notation:

$$\tilde{C} \cdot \tilde{X} > v(P) - v(\overline{P}) \qquad \text{since } \tilde{C}_j \geq 0, \quad j \in J. \tag{3}$$

Also, since every constraint is covered by exactly one variable in $\tilde{x}$, the following relationship must hold:

$$\tilde{U}A \cdot \tilde{X} = \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij}\tilde{u}_i \right) \tilde{x}_j = \sum_{i=1}^{m} \tilde{u}_i \,. \tag{4}$$

Then, by definition, we have:

$$\begin{aligned} \tilde{C} \cdot \tilde{X} &= (C - \tilde{U}A) \cdot \tilde{X} = C \cdot \tilde{X} - \tilde{U}A \cdot \tilde{X} \\ &= C \cdot \tilde{X} - \sum_{i=1}^{m} \tilde{u}_i \qquad \text{from (4)} \\ &= C \cdot \tilde{X} - v(D) \\ &> v(P) - v(\overline{P}) \qquad \text{from (3).} \end{aligned}$$

Since $v(D) = v(\overline{P})$, we get $C \cdot \tilde{X} > v(P)$ after cancellation. This is a contradiction, which means $\tilde{X}$ cannot be optimal. ∎

Theorem 1 implies that if we solve $\overline{P}$ optimally, we can eliminate all variables with reduced cost greater than the amount $v(P) - v(\overline{P})$ without loss of optimality. In general, v(P) is unknown and we substitute UB, the objective value of a known feasible solution to P. If no feasible solution is available, one can estimate the upper bound as $v(\bar{P}) + \delta$. (We will describe how δ may be determined shortly.)

If the above estimation is used, then P can be reduced by eliminating all variables with reduced cost greater than δ. We refer to the reduced problem as P_δ. Since P_δ is a restriction of P, $v(P_\delta) \geq v(P)$ and $F(P_\delta) \subseteq F(P)$. In fact, F(P) can be partitioned into two regions: $F(P) \cap \bar{F}(P_\delta)$ and $F(P_\delta)$. The following two theorems are direct extensions of Sweeney and Murphy's results.

Theorem 2: If $v(P_\delta) < v(\bar{P}) + \delta$, then $v(P_\delta) = v(P)$.

Proof : Omitted.

Theorem 3: If $v(P_\delta) > v(\bar{P}) + \delta$, then $v(P_\delta) - v(P) < v(P_\delta) - v(\bar{P}) - \delta$.

Proof : Omitted.

Implementing Variable Elimination

There are different strategies for implementing the techniques of variable elimination. We will suggest a strategy which can be employed after MAM is applied at step (1b) of the new algorithm. Suppose the optimal integer solution is not obtained by MAM. Then the $\tilde{C}_j$s can be estimated by the latest values of $SLACK_j$s, the slacks of the dual constraints in (2). Our experience indicates that the $SLACK_j$s are very good estimator of the $\tilde{C}_j$s because U is very close to $\tilde{U}$.

The success of variable elimination depends on how well the value of δ is selected. If δ is too large, only few variables are eliminated and the effect on the algorithm is negligible. On the other hand, if δ is too small, the optimal solution may be eliminated. We will discuss how to deal with the latter case shortly.

If the value of δ is chosen appropriately, a significant fraction of the variables can be eliminated without loss of optimality. This, in turn, can result in a tremendous saving in computation time for two reasons. First, the majority of the computational effort is devoted to evaluating the min expressions within MAM, and the effort required to evaluate these expressions is proportional to the cardinality of the J_is. (See the Appendix.) Since variable elimination decreases the cardinalities of the J_is, computation time is reduced also. Secondly, variable elimination increases the likelihood of fathoming a candidate subproblem due to infeasibility. When an infeasible subproblem is fathomed, we not only save the effort of applying MAM to the subproblem, but also reduce the size of the search tree by eliminating the need to perform further branching.

Our strategy not only implements variable elimination at the last iteration of MAM, but also within each iteration of MAM. We expect to gain significant saving in computation time even within each iteration of MAM. Since $SLACK_j$s are evaluated within each iteration of MAM, variable elimination can be applied with no extra effort. If $SLACK_j$ is greater than δ, then we simply remove variable j from $\bar{J}$.

Although the same value of δ can be used for each iteration, it is more effective to decrease δ at each iteration. The reason is that the accuracy of $SLACK_j$ as an estimator of $\tilde{C}_j$ improves at each iteration. To facilitate the discussion below, we introduce the following notation:

LB(t) = the value of LB obtained at iteration t

p(t) = LB(t) / v(P)

$\hat{p}(t)$ = an estimator of p(t) for future problems

$\delta(t)$ = the value of δ for iteration t

We now suggest a statistical method for choosing the $\delta(t)$ values. We have observed that the performance of MAM is fairly consistent when applied to problems originating from the same application. (By consistent, we mean that p(t) varies little among problems.) Suppose that we solve some problems from a specific application using the new algorithm BB-SPP and determine the values of p(t) for each problem. Then we can set $\hat{p}(t)$

to be the average of the p(t)s. If the variance of p(t) is large, then a conservative value, such as the minimum p(t), can be used instead.

In fact, for a specific application, a forecasting component can be incorporated into the algorithm. Each time BB-SPP is applied to a problem, the values of LB(t) can be saved. When the problem is solved and an optimal solution is obtained, then the values of p(t) can be calculated and used to update the $\hat{p}(t)$s. A forecasting method, such as exponential smoothing (Brown and Meyer, 1960), can be used for the updating.

With the values of $\hat{p}(t)$ available, we can then apply variable elimination to any future problem from the same application. At the end of each iteration t of MAM (during step 1b of BB-SPP), we simply set $\delta(t) = \{[1 / \hat{p}(t)] - 1\}\, LB(t)$ and delete any variables with $SLACK_j > \delta(t)$. At the termination of MAM, we apply the remaining steps of BB-SPP to the reduced problem, P_δ. When the optimal solution to P is obtained, we again update the $\hat{p}(t)$s as before.

To determine whether the optimal solution to P_δ is also optimal to P, we simply apply the results of Theorems 2 and 3. If $v(P_\delta) < LB + \delta$ (where LB and δ are the values obtained at the final iteration of MAM at step 1b of BB-SPP), then the solution to P_δ is also optimal to P. Otherwise, we need to increase the $\delta(t)$s and resolve the problem.

If the best incumbent obtained during the first trial is not optimal to P, then we can set $\delta(t) = v(P_\delta) - LB(t)$ during the second trial. These settings guarantee the solution obtained at the second trial will be optimal to P. However, if no incumbent was obtained during the first trial, we must either set the $\delta(t) = \infty$ or increase them by some step size and then resolve the new problem.

6. Computational Results & Discussion

Data & Results

All computational results are based on a set of ten crew scheduling problems from Chan and Yano. The size, density and optimal objective value of each problem are given in Table I. The number of constraints ranges from 65 to 88, and the number of variables ranges from 327 to 3919. For all problems, the densities are approximately 10% and the maximum value for $|I_j|$ is 17.

TABLE I

Description of Data Set.

Problem #	# of rows	# of columns	density (%)	v(P)
1	68	327	11.0	7112
2	70	656	9.6	7376
3	72	664	9.6	4823
4	65	827	10.4	7706
5	66	988	10.4	6664
6	88	990	8.6	7164
7	73	1321	9.9	5234
8	85	2161	9.3	6263
9	78	3839	10.5	7927
10	86	3919	10.3	5431

We applied BB-SPP to each of the 10 problems, permitting a maximum of five iterations for all applications of MAM. We found that five iterations were more than adequate to obtain fairly tight lower bounds. Furthermore, it was not necessary to perform a larger number of iterations at the root node as is done in most branch-and-bound algorithms. The results for BB-SPP are shown in Table II. In our application, the set-covering solution obtained at the end of each iteration of MAM does not constitute a valid upper bound for the SPP. Hence, UB is initially set to equal ∞ and updated only when an incumbent is found.

In Table II, four rows of information are provided for each problem. The first row, '# nodes', shows the total number of nodes created during the algorithm. These numbers include the root node and provide an indication of the size of the search trees. The second row, 'opt. @', gives the nodes at which each optimal solution was found. The third row, '#

TABLE II

Performance comparison in terms of total # of nodes investigated, node at which the optimal solution is found, total # of MAM iterations performed, and computation time†.

Prob. #		SETPAR	BB-SPP	BB-SPP2	BB-VE
1	# nodes	—	7	7	7
	opt. @	—	7	7	7
	# iter.	—	27	11	7
	time	0.067	0.043	0.026	0.018
2	# nodes	—	19	16	16
	opt. @	—	17	11	11
	# iter.	—	82	26	23
	time	0.106	0.236	0.085	0.047
3	# nodes	—	1	1	1
	opt. @	—	1	1	1
	# iter.	—	2	1	1
	time	0.140	0.019	0.013	0.013
4	# nodes	—	4	1	1
	opt. @	—	2	1	1
	# iter.	—	6	3	3
	time	0.220	0.046	0.029	0.026
5	# nodes	—	1	1	1
	opt. @	—	1	1	1
	# iter.	—	3	3	3
	time	0.220	0.031	0.031	0.025
6	# nodes	—	4	4	4
	opt. @	—	4	4	4
	# iter.	—	15	13	7
	time	0.327	0.113	0.109	0.046
7	# nodes	—	1	1	1
	opt. @	—	1	1	1
	# iter.	—	3	2	2
	time	0.366	0.047	0.038	0.034
8	# nodes	—	4	1	1
	opt. @	—	2	1	1
	# iter.	—	7	1	1
	time	0.620	0.137	0.041	0.041
9	# nodes	—	13	22	22
	opt. @	—	12	16	16
	# iter.	—	61	34	31
	time	2.384	1.761	0.876	0.185
10	# nodes	—	4	1	1
	opt. @	—	3	1	1
	# iter.	—	11	5	5
	time	2.574	0.485	0.278	0.166

†Computation times are given in seconds on an IBM 3090-200.

iter.', shows the total number of iterations of MAM performed. The fourth row, 'time', gives the computation times.

The times under the columns BB-SPP, BB-SPP2 and BB-VE are in seconds for an IBM 3090–200 under the Michigan Terminal System. (BB-SPP2 and BB-VE will be described shortly.) The times under the column SETPAR are also for an IBM 3090–200 but under the operating system at American Airlines. Computation times do not include the times for input and initialization. In all but problem 2, the computation times for BB-SPP are less than those for SETPAR. In fact, BB-SPP is 4.3 times faster than SETPAR on average, based on the ten problems.

This timing comparison is biased because SETPAR is a production code which has been improved over the last ten years, while BB-SPP is only an experimental code. Currently, BB-SPP is written in Pascal and is not optimized with respect to efficiency or storage. The only data structures used are arrays. Because of the extensive use of set operations in the algorithm, we expect the use of more efficient data structures will help to reduce the computation time of BB-SPP noticeably.

The results of BB-SPP also show that very few nodes were created for all the problems. Moreover, the optimal solutions were found at the end or close to the end of the computations (i.e., 'opt. found' is close to '# nodes'). This implies that little effort was required to verify the optimality of the solution, which is a very desirable characteristic of any branch-and-bound algorithm. Typically, branch-and-bound algorithms find the optimal solution relatively quickly, but a majority (i.e., as much as two-thirds or more) of the effort is spent verifying that the best incumbent is indeed optimal.

The success of BB-SPP is due to its ability to frequently select the *right* branch (i.e., containing the optimal solution) to investigate. Typically, only three candidate subproblems are created from each branching. When MAM is applied to the newly created subproblems, the one containing the optimal solution in its feasible region often possesses the lowest lower bound among all subproblems. Infrequently, the *wrong* branch may be selected. For these instances, the *breadth-first* policy (for selecting the next candidate problem) prevents the algorithm from spending excessive effort investigating this branch.

Next, we introduce three strategies which can easily be incorporated into the algorithm to improve its performance.

Improvement Strategies for BB-SPP

The three improvement strategies presented below are based on the notion that if the improvement in LB during an interation of MAM has not resulted in the fathoming of the current candidate subproblem, then the effort spent on the iteration is wasted.

This wasted effort can occur in three instances: 1) LB > v(P), but the current incumbent is not tight enough to fathom the candidate subproblem; 2) the current candidate subproblem is very difficult for MAM and LB increases very slowly; and 3) LB = v(P), but the optimal integer solution cannot be detected due to primal degeneracy. We now present three improvement strategies which attempt to alleviate the difficulties encountered in the three instances, respectively.

Let $\underline{P}$ denote the candidate problem with the smallest lower bound, $\underline{LB}$, among all the candidate problems in LIST. Then the first strategy is as follows: if LB > $\underline{LB}$ for the current iteration of MAM, then terminate MAM and store the current candidate subproblem in LIST. The logic is that if LB is greater than $\underline{LB}$, then there is a greater chance that LB is larger than v(P). Moreover, it is most likely that $\underline{P}$ contains the optimal solution to P.

Hence, by terminating MAM and storing the candidate subproblem, the algorithm is able to pursue the candidate problem $\underline{P}$ instead. Hopefully, when the candidate subproblem is reconsidered in the future, the incumbent will be tight enough then to fathom it and eliminate the need for further branching. This strategy is consistent with the breadth-first selection policy (employed in the branching strategy), which always retrieves candidate problem $\underline{P}$ for branching.

The second strategy is to terminate MAM and to store the candidate subproblem whenever the increase in LB during the current iteration is less than ρ, which we set to be 1.0. The reason for this strategy is that LB increases very slowly for difficult problems, especially as it approaches the optimal objective value of the LP relaxation. Hence, when the increase in LB is less than ρ, it is advantageous to store the candidate subproblem for the moment and pursue candidate problem $\underline{P}$ instead. When the candidate subproblem is retrieved in the future and branching occurs, each of the newly created candidate subproblems is likely to be less difficult, and LB should increase much more rapidly when MAM is applied to each of them. This strategy prevents wasting excessive effort on solving difficult candidate subproblems.

The third strategy is to try to detect when LB= v(P) but there is primal degeneracy, and then to apply a test to determine whether SOL contains an optimal solution to the candidate subproblem. When SOL is feasible, it is also optimal. (See the appendix.) However, when the optimal solution is primal degenerate, SOL may not be feasible. In this situation, MAM cycles and LB remains the same indefinitely. To avoid this problem, certain variables in SOL must be set to zero.

The third instance occurs when LB remains the same for two consecutive iterations. If all the C_j's are integers, it can also occurs when LB is an integer. Under these conditions, the following test can be applied: 1) identify each variable j in SOL which satisfies the condition $I_j \subseteq S_2$; 2) enumerate all combinations of the variables identified; 3) for each combination, set the variables in the selected combination to zero, and check whether the resultant solution is feasible. If a solution is feasible, then the candidate subproblem is fathomed and the incumbent may be updated. Since usually only a few variables are identified at step 1 above, the test actually takes little effort to perform.

We incorporated these three strategies into BB-SPP and refer to the new version as BB-SPP2. Computational results for BB-SPP2 are also shown in Table II. All problems, except problem 5, showed a noticeable improvement, particularly in the computational times. The application of strategies 1 and 2 resulted in a reduction in the total number of iterations required (e.g., problem 1, 2, 6, and 9). For problem 4, 8 and 10, the application of strategy 3 allowed the optimal solutions to be identified at the root nodes.

Implementation of Variable Elimination

We implemented variable elimination as described in section 4. First, we obtained the values of p(t) for the ten test problems. The values of p(t) indicate how close the LB(t)s are to v(P). These values are shown in Table III for iteration 1 to 5, and are given in percentages. The average over the ten problems for each iteration is given in the last column. We define $\underline{\delta}(t)$ to be the minimum $\delta(t)$ which can be used without eliminating the optimal solution. These values are also provided in Table III, but are normalized (i.e., divided by the corresponding v(P)) for ease of comparison.

For many of the problems, the normalized $\underline{\delta}(t)$ is less than $1 - p(t)$. This means that, for these problems, the value for $\delta(t)$ can be less than $v(P) - LB(t)$ without eliminating the optimal solution. The reason is that, at iteration t, $v(P) - LB(t) = \sum_{j=1}^{n} SLACK_j \tilde{x}_j$, where $\tilde{X}$ is the optimal solution. Hence, $\delta(t)$ only needs to be greater than $\max_{j:\tilde{x}_j=1} \{SLACK_j\}$. This implies that even when SOL is far from optimal, choosing a small value for $\delta(t)$ may not necessarily eliminate the optimal solution. This in turn results in robustness of the methodology. (For example, in problem 9, $p(1) = 87\%$ but $\delta(1)$ is only 5%.)

Suppose we assume the ten test problems represent a good sample for the application from which they originated. Then the p(t)s provide a basis for choosing the $\hat{p}(t)$s. Once the $\hat{p}(t)$s are chosen, they can be used to determine the $\delta(t)$s for future problems. However, since we do not have available to us any other problems originating from the same application as the test problems, we selected some conservative settings for the $\hat{p}(t)$s instead, and applied them to the ten test problems.

TABLE III

Statistics Relevant to Variable Elimination.

Iteration t		Problem 1	2	3	4	5	6	7	8	9	10	Average
1	$p(t) * 100$	95.07	94.61	100.0	99.35	98.03	95.36	99.30	100.0	87.14	95.81	96.47
	$[\underline{d}(t)/v(P)]*100$	4.43	4.30	0.00	0.65	1.12	2.77	0.70	0.00	5.00	3.14	2.21
	$[\bar{d}(t)/v(P)]*100$	9.40	9.36	9.89	9.83	9.70	9.43	9.82	9.89	8.62	9.48	9.54
	$[\lvert J_A\rvert / \lvert J\rvert]*100$	15.3	16.8	7.7	14.5	13.1	14.4	10.7	6.8	6.6	6.0	11.2
2	$p(t) * 100$	99.76	95.70	OPT	99.64	99.64	98.53	100.0	100.0	93.10	99.47	98.58
	$[\underline{d}(t)/v(P)]*100$	0.24	4.30	0.00	0.36	0.36	1.47	0.00	0.00	4.22	0.53	1.15
	$[\bar{d}(t)/v(P)]*100$	8.68	8.33	8.70	8.67	8.67	8.57	8.70	8.70	8.10	8.65	8.58
	$[\lvert J_A\rvert / \lvert J\rvert]*100$	14.1	14.6	6.3	9.9	11.5	11.0	8.4	5.4	5.9	4.7	9.2
3	$p(t) * 100$	99.88	96.02	—	100.0	OPT	99.15	OPT	100.0	94.67	99.52	98.92
	$[\underline{d}(t)/v(P)]*100$	0.12	3.85	0.00	0.00	—	0.85	—	0.00	3.75	0.41	1.12
	$[\bar{d}(t)/v(P)]*100$	7.52	7.23	7.53	7.53	—	7.47	—	7.53	7.13	7.49	7.43
	$[\lvert J_A\rvert / \lvert J\rvert]*100$	12.8	12.5	5.4	8.3	—	9.6	—	4.3	4.9	3.9	7.7
4	$p(t) * 100$	99.94	96.61	—	100.0	—	99.41	—	100.0	95.20	99.73	99.09
	$[\underline{d}(t)/v(P)]*100$	0.06	3.27	0.00	0.00	—	0.59	—	0.00	3.63	0.27	0.98
	$[\bar{d}(t)/v(P)]*100$	6.38	6.16	6.38	6.38	—	6.34	—	6.38	6.07	6.36	6.31
	$[\lvert J_A\rvert / \lvert J\rvert]*100$	10.7	11.3	4.7	6.8	—	7.7	—	3.5	3.8	3.0	6.4
5	$p(t) * 100$	99.97	97.17	—	OPT	—	99.65	—	OPT	95.63	OPT	99.24
	$[\underline{d}(t)/v(P)]*100$	0.03	2.73	0.00	0.00	—	0.35	—	0.00	3.60	0.00	0.84
	$[\bar{d}(t)/v(P)]*100$	5.26	5.11	5.26	5.26	—	5.24	—	5.26	5.03	5.26	5.21
	$[\lvert J_A\rvert / \lvert J\rvert]*100$	9.5	10.1	4.2	5.7	—	6.7	—	3.0	3.2	2.3	5.6

The settings selected are: $\hat{p}(1) = 0.91$, $\hat{p}(2) = 0.92$, $\hat{p}(3) = 0.93$, $\hat{p}(4) = 0.94$ and $\hat{p}(5) = 0.95$. The resultant $\delta(t)$s, also normalized by the corresponding v(P), are shown in Table III. Comparisons between the normalized $\delta(t)$s and the normalized $\underline{\delta}(t)$s indicate that the $\hat{p}(t)$s are indeed rather conservative estimates.

Nevertheless, the use of these conservative $\hat{p}(t)$s resulted in the elimination of a large number of variables, even after only one iteration of MAM. For each problem, the percentage of variables remaining at each iteration is also shown in Table III. On average, for the ten problems, only 11.2% of the variables remain after the first iteration, and 5.6% remain after five iterations. The results also indicate that, at a given iteration, the percentage remaining becomes smaller as the size of the problem increases. Such dramatic reduction of the number of variables, especially for the larger problems, will reduce the computational effort tremendously.

We denote the version with variable elimination and improvement strategies as BB-VE. The results for BB-VE are shown in Table II. In all but problem 3, the computation times for the problems are indeed reduced significantly. On average, based on the test problems, BB-VE is 9.5 times faster than SETPAR. This ratio increases as the size of the problem increases. Considering that SETPAR is a production code while BB-VE is only an experimental code, we conclude that BB-VE is significantly faster than SETPAR, at least for the test problems.

As mentioned in section 4, the aims of variable elimination are: to reduce computational effort of each iteration and to increase the likelihood of fathoming the subproblems. The first aim has obviously been achieved by the reduction in the computation times. The second aim has also been achieved in problem 1, 2, and 6, where the number of iterations were reduced due to infeasibility of certain subproblems. However, the numbers of nodes created remained the same.

To further improve the performance of the algorithm, we can apply variable elimination to each candidate subproblem as well. At each iteration t, we would simply eliminate variables with SLACK_j greater than UB − LB(t). This will reduce the size of each subproblem and increase the likelihood of fathoming it.

We feel that the problems available to us are neither large enough nor complicated enough to reveal the power of variable elimination. For more difficult problems (i.e., problems which result in larger numbers of nodes and require larger numbers of iterations of MAM), we expect to see not only reductions in computation time and number of iterations performed, but also reduction in the search tree.

7. Conclusion

We have introduced a new and effective branch-and-bound algorithm for the set partitioning problem (SPP), which outperforms SETPAR, the best algorithm currently available for the SPP. The new algorithm employs a lower bounding method called MAM, and a new branching strategy, which we developed to take advantage of the near-optimal solution provided by MAM.

This strategy proved to be superior to existing branching strategies for the SPP, and resulted in relatively small search trees. Typically, only three subproblems are created from a branching. When MAM is applied to the newly created subproblems, the one containing the optimal solution in its feasible region often possesses the lowest lower bound among all subproblems. Hence, in almost all cases, the algorithm selects the *correct* branch (i.e., containing the optimal solution) to investigate. As a result, relatively few nodes need to be examined.

We also presented three improvement strategies which are easily incorporated into the algorithm and require almost no extra effort. Results showed that these strategies reduced the number of iterations of MAM required and improved the computation times noticeably.

Finally, we extended the results of variable elimination to the SPP, and described how the techniques of variable elimination can be applied. Results for the test problems show that, on average, 94.4% of the variables can be eliminated without affecting the optimal solution. Elimination of the variables reduces the computational effort of MAM and increases the likelihood of fathoming the subproblems. When variable elimination and the three improvement strategies above were incorporated, the computation times were, on average, 9.5 times faster than SETPAR. Moreover, this ratio is observed to increase as the size of the problem increases.

Considering the rather low computation times required for solving the moderate-sized test problems, we expect the new algorithm to be superior to existing approaches for solving large-scale problems as well.

Appendix: Implementation of MAM

The following notation is relevant to MAM:

t = current iteration number

T = maximum number of iterations allowed

UP = $\{ i \in \bar{I} : u_i$ permitted to increase$\}$

$\text{SLACK}_j = C_j - \sum_{i \in I_j} u_i$

SOL = $\{ j \in \bar{J} : \text{SLACK}_j = 0 \}$

$\text{LB} = \sum_{i \in \bar{I}} u_i + \sum_{j \in F} C_j$

MODE = 1 or 2 (indicates mode of operation)

In this appendix, we will refer to the dual variables (u_is) as multipliers to avoid confusion with the integer variables (x_js). The constraints in (1) will be referred to as the primal constraints, while the constraints in (2) will be referred to as the dual constraints.

Algorithm

Step (0) :

```
    (a) If $S_0 = S_2 = \emptyset$, go to step 5.

    (b) Set $t = 1$.
```

Step (1) : Increase selected multipliers.

```
    (a) If $S_0 \neq \emptyset$, set UP = $S_0$ and MODE = 1.

        Otherwise, set UP = $\bigcup_{j \in SOL'} I_j \cap S_1$ and MODE = 2
           where $SOL' = \{ j \in SOL : I_j \cap S_1 \neq \emptyset,\ I_j \cap S_2 \neq \emptyset \}$.

    (b) If UP = $\emptyset$, terminate.

        Otherwise, for each $i \in UP$:

           set $u_i = u_i + \min_{j \in J_i \setminus SOL} \{ SLACK_j / | I_j \cap UP | \}$.
```

Step (2) : Decrease certain multipliers in violated dual constraints.

```
    If MODE = 2, then for each $j \in SOL'$ in turn:
            set $u_i = u_i + (n_i / \sum_{i \in I_j \cap S_2} n_i) * SLACK_j \quad i \in I_j \cap S_2$.
```
Step (3) : Ensure all primal constraints are covered.

```
    If MODE = 2, then repeat until $S_0 = \emptyset$:

        (a) Let $p = \min_{i \in S_0} \{ i \}$.
        (b) Set $u_p = u_p + \min_{j \in J_p} \{SLACK_j\}$.
        (c) Set $S_0 = S_0 \setminus I_q$ where $q = \arg\min_{j \in J_p} \{SLACK_j\}$.
```
Step (4) : Determine LB and check termination criteria.

```
    (a) If $LB \geq UB$, fathom.

    (b) If $S_0 = S_2 = \emptyset$, go to step 5.

    (c) Set $t = t + 1$. If $t = T$, terminate.

    (d) Go to step 1.
```
Step (5) : Check if incumbent should be updated.

```
    (a) If $LB < UB$, set $UB = LB$ and update incumbent.

    (b) Fathom.
```

Brief Description of MAM

When a candidate subproblem is created, certain variables are deleted from $\bar{J}$. This means that the contents of S_0 and/or S_2 may be altered. If both S_0 and S_2 are empty, then the current integer solution SOL is optimal to the subproblem. In this case, the algorithm goes directly to step 5 to check whether the incumbent should be updated; then the subproblem is fathomed.

At step (1a), certain multipliers are selected to be increased. These multipliers are selected with the aim of satisfying the primal constraints that are violated by the current SOL. The logic underlying the selection rules are given in Chan and Yano. If no multiplier is selected, then the algorithm terminates. This termination criterion usually occurs when $LB = v(D)$.

At step (1b), each selected multiplier is increased by an amount which ensures that at most one variable from J_j will enter the solution. Hence, when MODE = 1, no dual constraint is violated at the end of step 1. When MODE = 2, only the dual constraints in the set SOL' are violated. To correct the violations, certain multipliers are decreased at step 2.

After the decreases, if any primal constraint is still not covered, then a multiplier corresponding to an under-covered constraint is selected at step (3a) and increased at step (3b). The amount of increase ensures that no dual constraint is violated. This process is repeated until all primal constraints are either tight or over-covered.

As a result, a feasible dual solution is obtained at each iteration. This in turn provides a valid lower bound to the subproblem. In addition, SOL is a feasible solution to the corresponding set-covering problem. The value of this feasible solution provides a valid upper bound to v(P) for many practical problems such as vehicle routing.

At step 4, if LB is greater than UB, then the subproblem is fathomed immediately. If SOL is optimal (i.e., $S_0 = S_2 = \emptyset$) or the number of iterations exceeds the maximum allowed, then the algorithm goes to step 5. Otherwise, the algorithm returns to step 1 and repeats the above process.

References

Balas, E. and M.W. Padberg (1979), "Set Partitioning – A Survey," in N. Christofides, A. Mingozzi and P. Toth (editors), *Combinatorial Optimization*, Wiley, Chichester, England.

Balinski, M.L. and R.E. Quandt (1964), "On an Integer Program for a Delivery Problem," *Operations Research*, **12**, 300–304.

Brown, R.G. and R.F. Meyer (1960), "The Fundamental Theorem of Exponential Smoothing," *Operations Research*, **9**, 6, 673–685.

Chan, T.J. and C.A. Yano (1987), "Finding Lower Bounds to the Set Partitioning Problem via Multiplier Adjustment," Technical Report 87-OR-7, Department of Operations Research and Engineering Management, School of Engineering and Applied Sciences, Southern Methodist University.

Christofides, N. and S. Korman (1973), "A Computational Survey of Methods for the Set Covering Problem," Report No. 73/2, Imperial College of Science and Technology, April 1973.

Day, R.H. (1965), "On Optimal Extracting from a Multiple File Data Storage System : An Application of Integer Programming," *Operations Research*, **13**, 3, 489–494.

Etcheberry, J. (1977), "The Set Covering Problem : A New Implicit Enumeration Algorithm," *Operations Research*, **25**, 760–772.

Fisher, M.L. (1981), "The Lagrangian Relaxation Method for Solving Integer Programming Problems," *Management Science*, **27**, 1, 1–18.

Fisher, M.L. and P. Kedia (1986), "A Dual Algorithm for Large Scale Set Partitioning," Purdue University, Krannert Graduate School of Management, Working Paper No. 894.

Garfinkel, R.S. and G.L. Nemhauser (1969), "The Set Partitioning Problem : Set Covering with Equality Constraints," *Operations Research*, **17**, 848–856.

Garfinkel, R.S. and G.L. Nemhauser (1970), "Optimal Political Districting by Implicit Enumeration Techniques," *Management Science*, **16**, B495-B508.

Garfinkel, R.S. and G.L. Nemhauser (1972), "Optimal Set Covering : A Survey," in A. Geoffrion (editor), *Perspectives on Optimization*, Addison-Wesley, Reading, Massachusetts.

Geoffrion, A.M. (1974), "Lagrangian Relaxation for Integer Programming," *Mathematical Programming Study*, **2**, 82–114.

Gerbracht, R. (1978), "A New Algorithm for Very Large Crew Pairing Problems," 18th AGIFORS Symposium, Vancouver, British Columbia, Canada, September, 1978.

Lu, Ming-Te (1970), "A Computerized Airline Crew Scheduling System," Ph.D. Thesis, University of Minnesota.

Marsten, R.E. (1974), "An Algorithm for Large Set Partitioning Problems," *Management Science*, **20**, 779-787.

Marsten, R.E., M.R. Muller and C.L. Killion (1979), "Crew Planning at Flying Tiger: A Successful Application of Integer Programming," *Management Science*, **25**, 12, 1175-1183.

Marsten, R.E. and F. Shepardson (1981), "Exact Solution of Crew Scheduling Problems Using the Set partitioning Mode : Recent Successful Applications," *Networks*, **11**, 165-177.

Michaud, P. (1972), "Exact Implicit Enumeration Method for Solving the Set Partitioning Problem," *IBM Journal of Research and Development*, **16**, 573-578.

Pierce, J.F. (1968), "Application of Combinatorial Programming to a Class of All-Zero-One Integer Programming Problems," *Management Science*, **15**, 191-209.

Pierce, J.F. and J.S. Lasky (1973), "Improved Combinational Programming Algorithms for a Class of All Zero-One Integer Programming Problems," *Management Science*, **19**, 528-543.

Revelle, C., D. Marks and J.C. Liebman (1970), "An Analysis of Private and Public Sector Location Models," *Management Science*, **16**, 12, 692-707.

Root, J.G. (1964), "An Application of Symbolic Logic to a Selection Problem," *Operations Research*, **12**, 4, 519-526.

Salkin, H.M. (1975), *Integer Programming*, Addison-Wesley, Reading, Massachusetts.

Sandi, C. (1979), "Subgradient Optimization," in N. Christofides, A. Mingozzi and P. Toth (editors), *Combinatorial Optimization*, Wiley, Chichester, England.

Steinman, H. and R. Schwinn (1969), "Computational Experience with a Zero-One Programming Problem," *Operations Research*, **17**, 5, 917-920.

Sweeney, D.J. and R.A. Murphy (1979), "A Method of Decomposition for Integer Programs," *Operations Research*, **27**, 1128-1141.

Sweeney, D.J. and R.A. Murphy (1981), "Branch and Bound Methods for Multi-item Scheduling," *Operations Research*, **29**, 853-864.

Thuve, H. (1981), "Frequency Planning as a Set Partitioning Problem," *European Journal of Operational Research*, **6**, 29-37.

Valenta, J.R. (1969), "Capital Equipment Decisions : A Model for Optimal Systems Interfacing," M.S. Thesis, M.I.T., June 1969.

UNIVERSITY OF MICHIGAN
3 9015 04732 6320