

Solving the Generalized Assignment Problem: An Optimizing and Heuristic Approach

Robert M. Nauss

*College of Business Administration, University of Missouri-St. Louis, 8001 Natural Bridge Road,
St. Louis, Missouri 63121, USA
robert_nauss@umsl.edu*

The classical generalized assignment problem (GAP) may be stated as finding a minimum-cost assignment of tasks to agents such that each task is assigned to exactly one agent and such that each agent's resource capacity is honored. This NP-hard problem has applications that include job scheduling, routing, loading for flexible manufacturing systems, and facility location. Due to the difficulty in solving "hard" GAPs to optimality, most recent papers either describe heuristic methods for generating "good" solutions or, in the case of optimizing methods, computational results are limited to 500 to 1000 binary variables.

In this paper we describe a special purpose branch-and-bound algorithm that utilizes linear programming cuts, feasible-solution generators, Lagrangean relaxation, and subgradient optimization. We present computational results for solving "hard" problems with up to 3000 binary variables. An unanticipated benefit of the algorithm is its ability to generate good feasible solutions early in the process whose solution quality generally dominates the solutions generated by two recently published heuristics. Furthermore, the computation time required is often less than the time taken by the heuristics. Thus, we have an optimizing algorithm that can be used quite effectively as a heuristic when proof of optimality is not an absolute requirement.

(Integer Programming; Assignment Problems)

1. Introduction


The classical *generalized assignment problem* (GAP) has been the subject of numerous research papers since 1975. The problem may be stated as finding a minimum-cost assignment of tasks (or jobs) to agents (or machines) such that each task is assigned to exactly one agent and such that each agent's resource capacity is honored. GAP applications run the gamut from job sched-

uling (Balachandran 1972) to routing (Fisher and Jaikumar 1981) to loading for flexible manufacturing systems (Mazolla et al. 1989) to facility location (Ross and Soland 1977). An extensive review of applications and algorithms (both exact and heuristic) appears in Cattrysse and Van Wassenhove (1992). The interested reader is directed there for more information on the wide range of applications of the GAP.

In this paper we develop a branch-and-bound algorithm to solve the GAP. We use random problem generators as proposed by other researchers to obtain large “hard” problems and solve them to optimality. An unanticipated benefit of the algorithm is its ability to generate good feasible solutions early in the search process. These feasible solution values generally dominate the solution values found by competing heuristic approaches. Moreover, the *time* required to find a better solution is often less! Based on the computational results in this paper, the algorithm at hand has the best of both worlds: the ability to generate optimal solutions *and* the ability to find high-quality feasible solutions in times faster than competing heuristic approaches.

Consider a minimization problem. A heuristic approach to such a problem generally exhibits four characteristics. First, by definition the heuristic does not guarantee that an optimal solution will be generated. Second, a heuristic finds “good” solutions relatively quickly. Third, a measure of the solution quality is unavailable without a given or internally generated lower bound. Fourth, a time, iteration, or quality limit is generally invoked to halt the heuristic.

An optimizing approach has five contrasting characteristics. First, it guarantees that an optimal solution (if one exists) will be found. Second, good solutions may or may not be found early in the solution process (~~if~~ ^{namely} cutting-plane methods generate an optimal solution only at the final iteration). Third, bounds on the quality of a feasible solution are often known due to the calculation of a lower bound. Fourth, the generation of some solutions of a certain quality may be deferred or eliminated due to the availability of bounds on the (unknown) optimal solution value. For example, suppose we have a feasible solution with objective value $v > 0$. Then we can eliminate or defer the search for solutions with values greater than $v - e$ or kv when e and k are positive user-specified parameters. Fifth, time, iteration, or quality limits may be invoked to halt the optimizing approach. In such a case where these limits are met, the solution generated (if any) has not been proven to be optimal.

Our optimization algorithm for the GAP appears to fulfill the general advantages of a heuristic approach, while simultaneously allowing the user to generate optimal solutions if desired. 

Thus in a real-time operation, “good” feasible solutions may be obtained for related processes, while allowing continued search and generation of improved solutions (or an optimal solution).

The paper is organized as follows. Section 2 describes the mathematical formulation of the GAP. Section 3 discusses previous approaches and computational work. The solution methodologies employed in our algorithm are described in Section 4. Section 5 explains the branch-and-bound algorithm and Section 6 describes computational results. A concluding section gives ideas for future work and extensions.

2. Model Formulation

The GAP may be formulated as a 0 - 1 integer linear programming (ILP) model. Let n be the number of tasks to be assigned to m agents ($n \geq m$) and define $N = \{1, 2, \dots, n\}$. We define the requisite data elements as follows:

c_{ij} = cost of task j being assigned to agent i

r_{ij} = amount of resource required for task j by agent i

b_i = resource units available to agent i .

The decision variables are defined as:

$$x_{ij} = \begin{cases} 1, & \text{if task } j \text{ is assigned to agent } i \\ 0, & \text{if not.} \end{cases}$$

The 0-1 ILP model may then be written as:

$$(P) \quad \text{minimize} \quad \sum_{i=1}^m \sum_{j=1}^n c_{ij} x_{ij} \quad (1)$$

subject to:

$$\sum_{j=1}^n r_{ij} x_{ij} \leq b_i, \forall i \quad (2)$$

$$\sum_{i=1}^m x_{ij} = 1, \forall j \in N \quad (3)$$

$$x_{ij} = 0 \text{ or } 1, \forall i, j. \quad (4)$$

The objective function (1) sums the costs of the assignments, while constraint (2) enforces the resource limitation for each agent. Constraint (3) ~~assures~~ ^{me} ensures that each job is assigned to exactly one agent. We allow all data elements to be real (certain efficiencies follow if the data elements are

assumed to be integral).

3. Previous Computational Research

Since Ross and Soland's (1975) seminal paper on the GAP or (P) appeared, there has been a steady progression in published solution efficiency and in the size and difficulty of problems addressed. Initially, random problems were generated such that individual costs and resource usage were not correlated, leading to easily solved problems, where variables with small costs and small resources were often equal to 1, and those variables with large costs and resources were often 0 in an optimal solution. Martello and Toth (1981) devised a random problem generator (dubbed D) that inversely correlated individual costs and resources. These problems were more difficult to solve. More recently, Laguna et al. (1995) developed another random problem generator (dubbed E) with a natural-log-based inverse correlation that results in even more difficult problems.

Gavish and Pirkul (1991) addressed the multi-resource GAP and utilized with some success the melding of a heuristic and a Lagrangean-relaxation-based branch-and-bound algorithm. However, their test problems were restricted to problems with uncorrelated cost and resource coefficients. Upwards of 75% of their test problems were solved without branching since the heuristic's solution value was within 1 unit of the Lagrangean relaxation value (for integer-valued costs and resources).

More recent algorithmic and computational results also mix heuristic and optimizing approaches. Amini and Racer (1999) developed an efficient heuristic for smaller problems and undertook an extensive statistical experimental design and analysis to compare it to earlier optimizing approaches. Cattrysse et al. (1994) used a set-partitioning heuristic. Guignard and Zhu (1996) used a Lagrangean decomposition approach in order to generate heuristic solutions. Chu and Beasley (1997) devised a genetic-algorithmic approach to solve GAP problems heuristically. Savelsbergh (1997) specialized a general-purpose solver, MINTO, to solve GAP problems to optimality using a column-generation approach. Cattrysse et al. (1998) add lifted-cover inequalities that describe an approximation to the convex hull of the resource constraints, then use the LINDO branch-and-bound software to solve "easier" GAP problems with up to 500 variables. Finally, Park et al. (1998) developed a Lagrangian dual-based branch-and-bound algorithm where "hard" problems with up to 500 variables are solved to optimality.

4. Solution Methodology

The fact that problem (P) is NP-hard leads us to consider a special-purpose branch-and-bound algorithm for its solution. Numerous researchers over the past thirty years or more have successfully attacked a number of special-structure integer programming problems with this “divide and conquer” strategy. We follow suit.

For a minimization integer program (P) , the goal of branch~~and~~ ^{\uparrow} bound ^{\uparrow} is to prove that a valid objective-function lower bound (LB) is greater than or equal to a valid upper bound (UB). When and if this can be shown, we conclude that UB is the optimal solution value for (P) . Of course, the optimal solution value is rarely known at the outset, and so initially the UB may represent the value of a feasible solution (or be set to ∞). As improved feasible solutions are found, the UB correspondingly decreases in value until the optimal solution value is found and proven.

On the other hand, lower bounds on the optimal solution value are often calculated by using the concept of relaxation. For example, integrality restrictions of variables may be relaxed or certain constraints may be ignored. Alternatively, a Lagrangean relaxation may be implemented where certain constraints are removed and placed in the objective function by using appropriate dual multipliers.

Typically such relaxations admit an LB that initially rarely exceeds a given UB. At this point one has a choice between two options. One can tighten the relaxation, or one can separate the problem (P) into two or more subproblems in the hope that the lower bounds of *all* subproblems are greater than or equal to the UB. If a given subproblem’s LB is less than the UB (and the corresponding subproblem’s solution is not feasible for (P)), one either attempts to tighten its relaxation thus increasing its LB, or one must separate the subproblem into two or more smaller subproblems. Of course if the subproblem solution is feasible for (P) , UB is set to the value LB and the subproblem is discarded from further consideration. If an LB of a subproblem is greater than or equal to UB, then that subproblem may be discarded. This continues until all subproblem LBs are greater than or equal to UB. This is the essence of the branch-and-bound approach. We give a detailed accounting of our special-purpose branch-and-bound algorithm in Section 5. In the next subsections we describe a number of procedures that attempt to increase the LB of (P) . These procedures may also be used to increase the lower bounds of subproblems of (P) .

4.1^⓪ Increasing the Lower Bound of (P) Using Linear Program Cuts

The linear programming relaxation of (P), namely (\bar{P}) , simply replaces constraint (4) with the constraint $x_{ij} \geq 0, \forall i, j$. We assume that the optimal solution of (\bar{P}) , \bar{x} , is not integer feasible. The linear program solution \bar{x} is examined, valid cuts are added, and the augmented linear program (\bar{P}') with cuts appended is then resolved. The new solution \bar{x}' is examined and valid cuts that are violated by \bar{x}' are added. This procedure continues to cycle until the number of cuts appended is greater than some user-specified maximum number or until no further cuts can be added that are violated by the current LP solution. Clearly we have $v(\bar{P}') \geq v(\bar{P})$ since valid violated constraints have been added to the linear program.

The type of cut that is used in this application is a minimal cover cut (Balas and Jeroslow 1972). These cuts are stronger than standard LP-based cuts since they take into consideration the structure of the polytope associated with the problem and are facet-defining inequalities for a lower-dimensional polytope. These cuts are developed as follows. Find a constraint i of type

(2), namely $\sum_{j=1}^n r_{ij} x_{ij} \leq b_i$, that has at least one \bar{x}_{ij} fractional. Let S_i be the index set of $\bar{x}_{ij} > 0$.

From the index set S_i , choose variables in decreasing order of r_{ij} and place the indices in set

F_i until $\sum_{j \in F_i} r_{ij} > b_i$. Let $C_i = |F_i| - 1$. Then the cut, $\sum_{j \in F_i} x_{ij} \leq C_i$, is a valid cut that the solution \bar{x}

violates as long as $\sum_{j \in F_i} \bar{x}_{ij} > C_i$. If $\sum_{j \in F_i} \bar{x}_{ij} \leq C_i$, then the cut is not added even though it is a valid

cut. If the cut is added, then strengthen the cut in the following way. Let $R = \max_{j \in F_i} \{r_{ij}\}$. Then for

all $j \in N - F_i$ and for which $r_{ij} \geq R$, add j to the set F_i (note that the value C_i remains unchanged). Finally we apply a coefficient-lifting algorithm to the cut using a procedure outlined in Cattrysse et al. (1998). This lifted cut is represented as follows: $\sum_{j \in F_i} a_{ij} x_{ij} \leq C_i$ where the a_{ij} 's

are positive and integer-valued.

4.2^⓪ Increasing the Lower Bound of (P) Using Lagrangean Relaxation

When no further cuts of the kind described above may be added, we proceed to tighten the current LP relaxation (\bar{P}') using Lagrangean relaxation. While at most one cut per resource con-

straint is added per cycle in the previous subsection, the repeated solving of an LP, adding cut(s), solving the LP with cuts appended, etc., means that multiple cuts on a given resource constraint may be present. Let $NC(i)$ be the total number of cuts added for resource constraint i and let

$\sum_{i=1}^m NC(i) = NC$ where NC is the total number of cuts added over all resource constraints. Then the

k th added cut for resource constraint i will be denoted as $\sum_{j \in F_{i(k)}} a_{i(k)j} x_{i(k)j} \leq C_{i(k)}$. The correspond-

ing dual variable of the cut $i(k)$ is $\gamma_{i(k)}$.

We have the following Lagrangean relaxation (LGR):

$$LGR(\lambda, \gamma) \equiv \text{minimize} \sum_{i=1}^m \sum_{j=1}^n c_{ij} x_{ij} + \sum_{j=1}^n \lambda_j \left(1 - \sum_{i=1}^m x_{ij} \right) + \sum_{i=1}^m \left[\sum_{k=1}^{NC(i)} \gamma_{i(k)} \left(C_{i(k)} - \sum_{j \in F_{i(k)}} a_{i(k)j} x_{i(k)j} \right) \right] \quad (5)$$

subject to:

$$\sum_{j=1}^n r_{ij} x_{ij} \leq b_i, \quad \forall i$$

$$x_{ij} = 0 \text{ or } 1, \quad \forall i, j.$$

*Ans: Wording OK?
Cannot find
in dictionary.*

(6)

(7)

As is evident, the multiple choice constraints (3) and the NC added cuts are Lagrangeanized. We note that $LGR(\lambda, \gamma)$ separates into m independent 0-1 knapsack problems. We use a special-[?] purpose branch-and-bound algorithm due to Nauss (1976) to solve the knapsacks. This was done because this algorithm allows real values for the objective and resource coefficients as opposed to requiring all-integer data.

The dual multiplier vectors λ and γ are obtained from the optimal LP solution to (\bar{P}') . The use of these dual multipliers ^{en?} assures that $v(LGR(\lambda, \gamma)) \geq v(\bar{P}')$ (see Geoffrion 1974). We then proceed to tighten the LGR further through the use of subgradient optimization.

Subgradient optimization attempts to solve:

$$\begin{aligned} &\text{maximize} \quad \left\{ \text{minimize} \sum_{i=1}^m \sum_{j=1}^n c_{ij} x_{ij} + \sum_{j=1}^n \lambda_j \left(1 - \sum_{i=1}^m x_{ij} \right) + \sum_{i=1}^m \left[\sum_{k=1}^{NC(i)} \gamma_{i(k)} \left(C_{i(k)} - \sum_{j \in F_{i(k)}} a_{i(k)j} x_{i(k)j} \right) \right] \right\} \quad (8) \\ &\lambda \text{ unrestricted} \\ &\gamma \leq 0 \end{aligned}$$

$$\text{subject to:} \quad \sum_{j=1}^n r_{ij} x_{ij} \leq b_i, \quad \forall i \quad (9)$$

$$x_{ij} = 0 \text{ or } 1, \forall i, j. \quad (10)$$

Note that λ is unrestricted since the corresponding constraint is an equation and that $\gamma \geq 0$ since the cut constraints are less-than-or-equal-to constraints. We limit the number of subgradient iterations to 100 and utilize a stepsize calculation as described in Guignard and Rosenwein (1989).

4.3 Increasing the Lower Bound of (P) Using Penalties

The lower bound of (P) may also be improved by showing that a variable x_{ij} must take on a particular value (0 or 1) in an optimal solution. This may be accomplished for example, through the use of penalties. If we can show that a relaxation where, say, $x_{ij} = 0$ has an objective value greater than or equal to the current UB, then we may “fix” or “peg” x_{ij} to the value of 1. We proceed to describe a penalty calculation using the Lagrangean relaxation from Section 4.2.

Consider the Lagrangean relaxation $LGR(\lambda, \gamma)$ where (λ, γ) are the dual multiplier vectors found via subgradient optimization. Note that $v(LGR(\lambda, \gamma))$ may be divided into $m + 1$ components. That is,

$$\left(\sum_{j=1}^n \lambda_j + \sum_{i=1}^m \sum_{k=1}^{NC(i)} \gamma_{i(k)} C_{i(k)} \right) + \sum_{i=1}^m \left(\sum_{j=1}^n c_{ij} - \lambda_j - \sum_{k=1}^{NC(i)} \gamma_{i(k)} \left(\sum_{j \in F_{i(k)}} a_{i(k)j} \right) \right) x_{ij}.$$

Defining

$$v(KNAP_i) = \sum_{j=1}^n \left(c_{ij} - \lambda_j - \sum_{k=1}^{NC(i)} \gamma_{i(k)} \left(\sum_{j \in F_{i(k)}} a_{i(k)j} \right) \right) x_{ij},$$

we have

$$v(LGR(\lambda, \gamma)) = \sum_{j=1}^n \lambda_j + \sum_{i=1}^m \sum_{k=1}^{NC(i)} \gamma_{i(k)} C_{i(k)} + \sum_{i=1}^m v(KNAP_i).$$

If we let $v(\overline{KNAP_i})$ be the optimal LP relaxation solution value for $(KNAP_i)$, we have for some i^* ,

$$v(LGR(\lambda, \gamma)) \geq \sum_{j=1}^n \lambda_j + \sum_{i=1}^m \sum_{k=1}^{NC(i)} \gamma_{i(k)} C_{i(k)} + \sum_{\substack{i=1 \\ i \neq i^*}}^m v(KNAP_i) + v(\overline{KNAP_{i^*}}).$$

Then the right-hand side of the above inequality is a valid relaxation value for $v(LGR(\lambda, \gamma))$. It is well-known that the LP penalties for variables in a knapsack problem are easily calculated using

the optimal LP dual multiplier for the resource constraint (see Nauss 1976 for more detail). Note also that the LP penalty for a variable x_{ij} is based on the LP relaxation of a *single* constraint while the remainder of the knapsack constraints are not relaxed. Thus if Z^* is the best solution to (P) known to date, we may peg $x_{i^*,j}$ to 1 if

$$\sum_{j=1}^n \lambda_j + \sum_{i=1}^m \sum_{k=1}^{NC(i)} \gamma_{i(k)} C_{i(k)} + \left(\sum_{\substack{i=1 \\ i \neq i^*}}^m v(KNAP_i) \right) + v(\overline{KNAP}_{i^*}) + PEN(i^*, j | x_{i^*,j} = 0) \geq Z^*. \quad (11)$$

In (11), $PEN(i^*, j | x_{i^*,j} = 0) = -c_{i^*,j} + \zeta_{i^*,j}$ where ζ is the optimal dual multiplier of (\overline{KNAP}_{i^*}) . We note further that an enhanced penalty may be calculated by finding $v(KNAP_{i^*} | x_{i^*,j} = 0)$ and substituting it for $v(\overline{KNAP}_{i^*}) + PEN(i^*, j | x_{i^*,j} = 0)$ in (11). This of course requires solving a 0-1 knapsack, which may be prohibitive if every variable were so tested. However, the use of such a “true” penalty can be effective when the left side of inequality (11) is “close” to Z^* . Finally we remark that pegs to 0 are handled analogously.

4.4⁶ Increasing the Lower Bound of (P) Using Feasibility-Based Tests

We now turn to feasibility-based tests for increasing the lower bound of (P) . We begin with a procedure that places upper bounds on the slack variables in each knapsack constraint. Rewrite

$\sum_{j=1}^n r_{ij} x_{ij} \leq b_i \quad i=1, \dots, m$ by adding a nonnegative slack variable, u_i . We have

$\sum_{j=1}^n r_{ij} x_{ij} + u_i = b_i \quad i=1, \dots, m$. Now solve the following linear program for each i in turn, where

Z^* is an upper bound on (P) :

$$g_i \equiv \text{maximize } u_i \quad (12)$$

subject to:

$$\sum_{j=1}^n r_{ij} x_{ij} + u_i = b_i, \quad \forall i \quad (13)$$

$$\sum_{i=1}^m x_{ij} = 1, \quad \forall j \quad (14)$$

$$\sum_{i=1}^m \sum_{j=1}^n c_{ij} x_{ij} \leq Z^* \quad (15)$$

$$x_{ij} \geq 0, \quad \forall i, j \quad (16)$$

$$u_i \geq 0, \quad \forall i. \quad (17)$$

Then each resource constraint (2) in (P) may be rewritten as $\sum_{j=1}^n r_{ij} x_{ij} + u_i = b_i$ with upper bounds

on the u_i 's depicted as $0 \leq u_i \leq g_i, \quad \forall i$.

Next we solve the following LP:

$$Z_u^* \equiv \text{maximize} \sum_{i=1}^m u_i$$

subject to:

$$(13) - (16)$$

$$0 \leq u_i \leq g_i, \quad \forall i.$$

Now suppose that for a particular subproblem a number of x_{ij} variables have been set to 1, and

that for some resource constraint ℓ we have: $\sum_{j|x_{ij}=1} r_{ij} x_{ij} > b_\ell - g_\ell$ and $\sum_{j|x_{ij}=1} r_{ij} x_{ij} \leq b_\ell$. Now for

all i define $FIXUND(i) = \min \left\{ b_i - \sum_{j|x_{ij}=1} r_{ij} x_{ij}, g_i \right\}$. Then $FIXUND(i)$ is the maximum slack that knap-

sack i may have. Then we know that $Z_u^* - \sum_{\ell=1, \ell \neq i}^m FIXUND(\ell)$ is an upper bound, $UBU(i)$, on the

maximum slack for knapsack i and that if $UBU(i) < g_i$, then g_i may be reduced to the value $UBU(i)$ for all descendant subproblems of the subproblem under consideration.

Another method for tightening $UBU(i)$ for a subproblem is to use LP penalties for each knapsack in the LGR . Knapsack LPs can be solved analytically (see Nauss 1976). Define \bar{u}_i to be the value of the slack variable in the i th knapsack LP. If $\bar{u}_i = 0$ in an optimal LP solution, it is straightforward to calculate the maximum value that u_i could assume such that the objective value of the LGR still exceeds Z^* . These calculations are carried out after all knapsacks in the LGR have been solved, since the test depends on $v(LGR_{\lambda, \gamma})$, $v(KNAP_i)$, and $v(\overline{KNAP_i})$. The

maximum value for u_i may then replace g_i in the current subproblem and all its descendants.

It should be noted that this tightening of the bound on u_i is often more useful in feasibility-based pegging tests than in immediately tightening the *LGR*. This is because the variable to be pegged to 0 is often equal to 0 in the *LGR*. The benefit of pegging the variable to 0 is important because this may eventually lead to other variables being pegged to 1.

4.5²⁰ Increasing the Lower Bound of (P) Using Logical Feasibility Tests

Logical feasibility tests may be used to “peg” variables to 0 or 1. That is, if one can show that a problem (or subproblem) is infeasible if x_{ij} takes on the value 1, then it is clear that x_{ij} may be fixed to 0. We proceed to describe three logical feasibility tests that may be used to peg x_{ij} variables.

The test
First is a conditional feasibility pegging test to peg some x_{ik} to 1. Assume x_{ik} is temporarily set to 0. Then if we can show that for some i , $\sum_{j|x_{ij}=1} r_{ij} < b_i - g_i$ and $\sum_{\substack{\ell \neq k \text{ and} \\ \ell \neq j | x_{ij}=1}} r_{i\ell} < b_i - g_i - \sum_{j|x_{ij}=1} r_{ij}$, then

x_{ik} may be pegged to 1. This follows since we must have $b_i - g_i \leq \sum_j r_{ij} x_{ij} \leq b_i$.

The test
Second is a conditional pegging test where x_{ik} is temporarily set to 1. If it can be shown that no feasible solution exists under this assumption, then x_{ik} may be pegged to 0. For example, if for some i , $\sum_{\substack{j \neq k \text{ and} \\ j | x_{ij}=1}} r_{ij} + r_{ik} > b_i$, then x_{ik} may be pegged to 0.

The test is as follows:

Third if for some i , $\sum_{j|x_{ij}=1} r_{ij} + r_{ik} < b_i - g_i$, and $\sum_{j|x_{ij}=1} r_{ij} - r_{ik} - \min_{\substack{\ell \neq k \text{ and} \\ \ell \neq j | x_{ij}=1}} r_{\ell} > b_i$, then x_{ik} may be

pegged to 0. Note that the first and third of these three feasibility tests depend on the upper bound g_i . When g_i can be reduced, the feasibility tests become stronger.

4.6^{*} Decreasing the Upper Bound for (P) Using Feasible-Solution Generators

Some recent research on the GAP has concerned the development of efficient heuristics, which may be used before invoking a branch-and-bound algorithm. In general, “good” feasible solutions are generated in a reasonable amount of CPU time. For this reason, we use the tabu-search code of Laguna et al. (1995) to generate an initial feasible solution before invoking the branch-...

and-bound algorithm.

Once the branch-and-bound algorithm has been invoked, we utilize other approaches to generate improved feasible solutions. First consider a Lagrangean relaxation solution \hat{x} from (8)-(10). Since the Lagrangean relaxation is solved with the multiple choice constraints (3) relaxed, it is often the case that these constraints are violated. For example, a job ℓ may have $\sum_{i=1}^m \hat{x}_{i\ell} \geq 2$ while another job k may have $\sum_{i=1}^m \hat{x}_{ik} = 0$. Typically, most multiple choice constraints have $\sum_{i=1}^m \hat{x}_{ij} = 1$. Accordingly, we attempt to “massage” the *LGR* solution, \hat{x} , by first attempting to reduce assignments as cheaply as possible for constraints where $\sum_{i=1}^m \hat{x}_{ij} \geq 2$. This is accomplished by finding the constraint j^* where $j^* = \arg \max \left(\sum_{i=1}^m \hat{x}_{ij} \right)$. Then we reset all but one \hat{x}_{ij} that are currently equal to 1 to the value 0 in nonincreasing order of the cost coefficients, c_{ij^*} . These assignment reductions create more slack in the resource constraints allowing possible assignments to be made for jobs where $\sum_{i=1}^m \hat{x}_{ij} = 0$. For each such constraint j^* we attempt to set an \hat{x}_{ij^*} to 1 where two conditions must be met. First the knapsack equation (9) must have sufficient slack to allow r_{ij^*} to fit *and* the cost coefficient c_{ij^*} should be as small as possible. If all multiple choice constraints are made equal to 1 *and* the objective function has value less than Z^* , an improved feasible solution has been found.

If and when an improved feasible solution has been found, we invoke a neighborhood slide/switch heuristic formalized by Ronen (1992) in order to find an even better solution. Briefly, Ronen attempts to move each job from its current agent to every other agent. The move is made only if a cost savings is achieved. Next, all pairs of jobs (j_1, j_2) that are assigned to distinct agents (i_1, i_2) where $i_1 \neq i_2$ are tentatively switched so that job j_1 is assigned to agent i_2 and job j_2 is assigned to agent i_1 . Once again the switch is made only if a feasible cost savings is achieved. Tentative switches of one job for two jobs with a different agent as well as 2 for 2 job switches and 2 for 3 job switches are also tested. Switches are made only if the resulting as-

signment satisfies the resource constraints *and* cost savings are achieved. If such an improved solution is formed, the entire neighborhood slide/switch and multiple switches are tried again until a complete cycle is completed without an improved feasible solution being found.

Another useful tool is complete enumeration when relatively few jobs (say, five or fewer) remain to be assigned in some subproblem. Every possible combination of assignments of jobs to agents is evaluated. If an improved feasible solution is found, the incumbent and its value Z^* is updated. In any case the subproblem is discarded from further consideration.

4.7⁰ Solution Strategies

We conclude Section 4 with an exhibit that shows the usage of the various solution methodologies in our algorithm.

1 ch.
. 25

Methodology/ Subsection	Purpose	Problem (P) Usage	Subproblem Usage	Computation Effort
LP Cuts (4.1)	Increase LB	Yes	No	High
#> LGR (4.2)	Increase LB	Yes	Periodic	High
#> Penalties (4.3)	Increase LB	Yes	All	Low
#> Feasibility-based Test (4.4)	Increase LB	Yes	Only when im- proved UB found	High
#> Logical Feasibility Test (4.5)	Increase LB	Yes	All	Low
#> Feasible Solution Gen- erator (Laguna) (4.6)	Decrease UB	Yes	No	High
#> Feasible Solution Gen- erator (other) (4.6)	Decrease UB	Yes	All	Medium
Complete Enumeration (4.6)	Decrease UB	No	5 or fewer free jobs	Low

Rule 2

5. Branch-and-Bound Algorithm

We now turn to a brief description of the branch-and-bound algorithm using the basic framework found in Geoffrion and Marsten (1972). A general flow chart of the algorithm is depicted in Figure 1. Detailed comments follow later in this section.

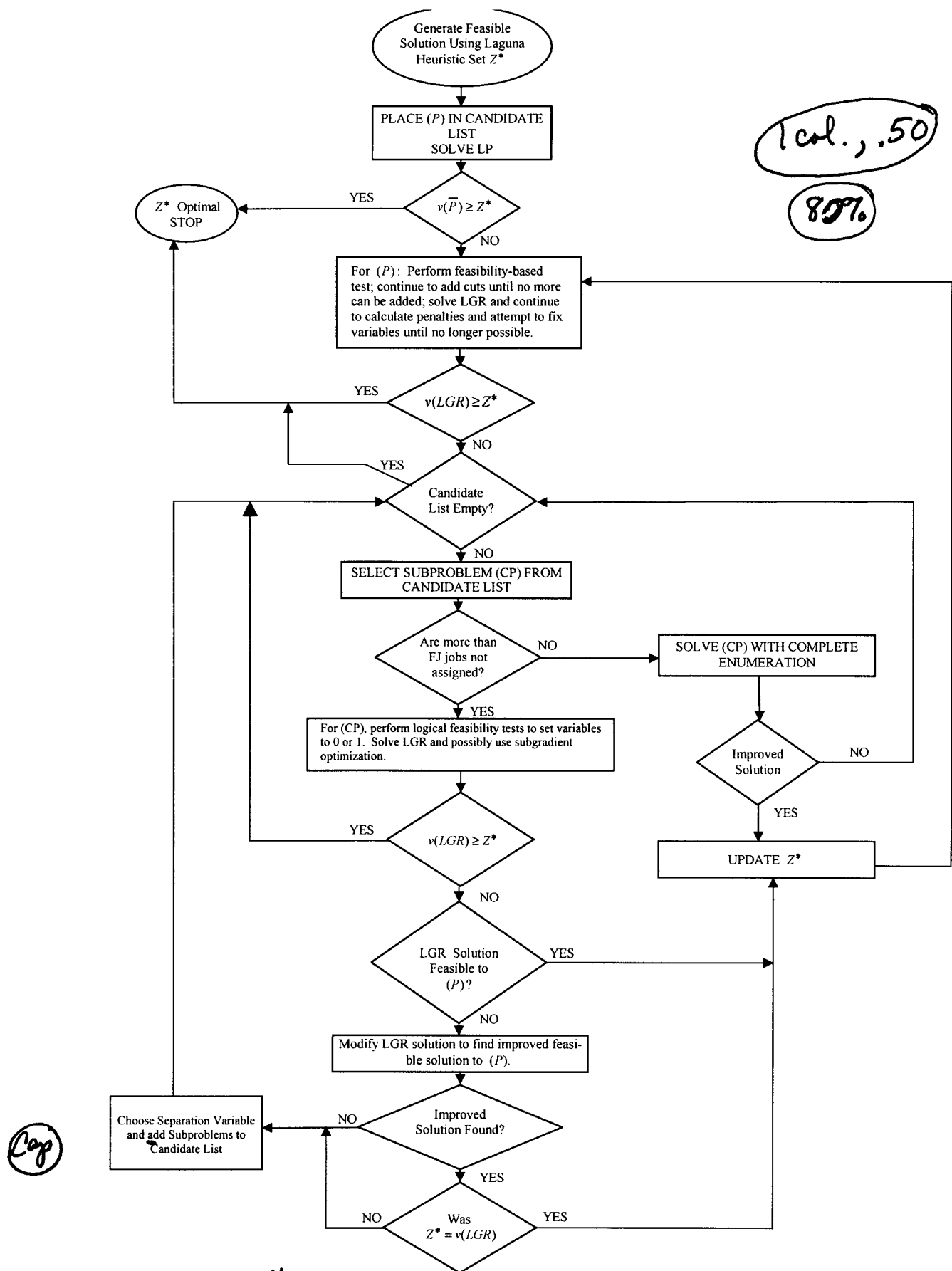


Figure 11 General Flow Chart for Branch-and-Bound Algorithm

Numbers roman three

Step 1. Initial feasible solution generator Utilize the tabu-search heuristic of Laguna et al. (1995) for ten seconds to obtain a feasible solution to (P) with objective value Z^* . If none is found, set $Z^* = \infty$.

Step 0. Set up Solve the linear programming relaxation (\bar{P}) . If $v(\bar{P}) \geq Z^*$, the incumbent solution is optimal; stop. Otherwise, place (P) in the candidate list. Set parameters as follows: FJ = 5, MAXCUT = 60, START = 10, EVERY = 5, CHT = 1.014, NODE = IBACK = FAKE = 0 (detailed descriptions of the parameters follow the algorithmic description).

Step 1. Preprocessing using improved feasible solution to (P) Solve $m+1$ linear programs to determine the maximum slack allowed for each resource constraint and the maximum total slack summed over all resource constraints as detailed in Section 4.4. Add cuts to (\bar{P}) as described in Section 4.1 and reoptimize (\bar{P}') with cuts appended. Using the new LP solution \bar{x}' add more cuts and reoptimize (\bar{P}') with the new cuts appended. Continue this procedure until no more cuts are possible or until MAXCUT cuts have been added. Using the dual multipliers from the last (\bar{P}') with cuts appended, solve the Lagrangean relaxation. If NODE = 0 and if $CHT * v(LGR) < Z^*$, set $Z^* = CHT * v(LGR)$, and set FAKE = 1. Apply subgradient optimization (see Section 4.2) in order to tighten the Lagrangean relaxation. If $v(LGR) \geq Z^*$, set IBACK = 1, and go to step 3.

Step 2. (Penalties) Attempt to fix variables to 0 or 1 using LP knapsack penalties. If NODE is a multiple of EVERY, attempt to fix variables with strong IP knapsack penalties by solving the associated knapsack problem (see Section 4.3). Attempt to fix variables to 0 or 1 using feasibility tests (Section 4.4). If any variables have been fixed to 0 or 1, resolve the Lagrangean relaxation and apply subgradient optimization (Section 4.2). If no variables have been fixed and IBACK = 1, go to step 3. If no variables have been fixed and IBACK = 0, go to step 8.

Step 3. (Candidate selection) Select a problem (CP) from the candidate list using LIFO, set NODE = NODE + 1, and set IBACK = 0. If the candidate list is empty, stop; an optimal solution has been found if FAKE = 0. If the candidate list is empty and FAKE

$\neq 0$ the value of CHT should be doubled and the algorithm restarted at Step 0.

- Step 4. (Enumeration) If (CP) has more than FJ jobs not assigned, go to step 5. Otherwise, solve (CP) with complete enumeration. Set IBACK = 1. If an improved feasible solution has been found, update Z^* , set FAKE = 0 and go to step 1. ~~Else~~ go to step 3.
- Step 5. Perform feasibility-based tests to fix variables to 0 or 1 (Section 4.5).
- Step 6. (Solve relaxation) Solve the Lagrangean relaxation of (CP). If $v(LGR) \geq Z^*$, set IBACK = 1 and go to step 3. If $NODE \leq START$ or NODE is a multiple of EVERY, use subgradient optimization to tighten the Lagrangean relaxation of (CP), and if $v(LGR) \geq Z^*$, set IBACK = 1, and go to step 3. If the Lagrangean relaxation solution is feasible for (P) update Z^* , set FAKE = 0, set IBACK = 1, and go to step 1.
- Step 7. ~~Feasible solution generator~~ Attempt to modify the optimal Lagrangean relaxation solution to find an improved feasible solution to (P) (see Section 4.6). If an improved feasible solution is found, update Z^* and set FAKE = 0. If $Z^* = v(LGR)$, set IBACK = 1 and go to step 1. If $Z^* > v(LGR)$, then fathoming may not occur, so go to step 8.
- Step 8. (Problem separation) Choose a free variable x_{ij} in (CP) and add the subproblems $(CP|x_{ij} = 0)$ and $(CP|x_{ij} = 1)$ to the candidate list. Go to step 3.

The parameter settings given in step 0 may be described as follows. NODE is simply a counter. The value START specifies that subgradient optimization in step 6 is to be invoked as long as $NODE \leq START$. The parameter EVERY = 5 refers to the frequency with which subgradient optimization is undertaken in step 6 and the frequency with which 0-1 knapsacks are solved in calculating strong IP penalties in step 2. The setting of FJ = 5 means that any candidate problem with FJ or fewer jobs remaining to be assigned is solved by complete enumeration as shown in step 4. MAXCUT = 60 refers to the maximum total number of cuts that may be added to the LP in step 1. The indicator IBACK is set to 1 if fathoming occurs. When IBACK = 0, then separation (or subproblem creation) in step 8 must occur. The indicator FAKE is initially set to 0. However, if the initial value of the (LGR) at NODE 0 (before subgradient optimization) is such that $CHT * v(LGR)$ is less than Z^* , then we set FAKE = 1 and replace Z^* by $CHT * v(LGR)$. The reason for this “cheating” is that the gaps between the (LGR) and the

best Z^* eventually found are very small (typically less than 1.4%). By initially setting Z^* to the smaller value we gain in two ways. First as mentioned in Guignard and Rosenwein (1989) it is helpful if a good upper bound target is used for subgradient optimization so that “overshooting” is less of a problem. Second, it may be possible that the use of this “cheating” Z^* may result in binary variables being fixed to 0 or 1 as a result of our penalty tests. We mention in passing that the Z^* set to $CHT * v(LGR)$ resulted in improved values of Z^* being found in all 684 problems solved in Section 6.

The values for FJ, START, and EVERY specified in step 0 of the algorithm were developed by comparing computational results for a small collection of test problems as each parameter was varied individually. No attempt was made to vary these three parameter settings simultaneously. The value of MAXCUT was determined so that associated storage requirements for the linear programming code remained reasonable. Finally, the value of CHT was determined after all 684 test problems were run initially and an analysis was made of the gaps for these problems.

In step 8 we determine the branch or separation variable to be used in creating two new candidate problems. Since the “1” branch is investigated initially (due to a LIFO candidate selection rule), we wish to select an x_{ij} variable with a strong tendency to take on the value 1. This is achieved as follows. Let J_{FIX1} be the index set of all jobs j that have an x_{ij} set (or fixed) to 1 in the current candidate problem. Then over all $j \notin J_{FIX1}$, place j in the index set JB if $\sum_{i=1}^m \hat{x}_{ij} = 1$ (where \hat{x} is the current Lagrangean relaxation solution). The branch variable x_{ij} is selected by finding $\max_{\substack{j \in JB \\ \text{and } \hat{x}_{ij}=1}} PEN(i, j | x_{i*j} = 0)$ where $PEN(*)$ is the knapsack LP penalty. In effect we select a variable x_{ij} with the largest penalty such that the corresponding multiple choice constraint $\left(\sum_{i=1}^m x_{ij} = 1 \right)$ is not violated in the Lagrangean relaxation.

6. Computational Results

The algorithm was coded in FORTRAN and compiled using Watcom FORTRAN 77 v10.5 with an optimization code OX (Watcom 1995). XMP (Marsten 1980) was used to solve the linear programs. A Dell XPS D300 (300 MHz Pentium II with 64 MB of SDRAM) was used for most

of the computational work.

Two datasets were used. First, 24 problems of types A, B, C, ~~D~~^{and} and of size ranging from 500 to ~~4,000~~^{3,000} binary variables were obtained from the OR Library maintained by Beasley and reported in Chu and Beasley (1997). A second dataset consisting of 640 random problems was generated according to specific guidelines. These guidelines are given in Chu and Beasley (1997) and in Laguna et al. (1995). The second dataset consists entirely of “hard” GAP problems of types D and E ranging from 500 to ~~3,000~~^{3,000} binary variables (see Section 3).

We present the computational results in seven tables. Table 1 reports on each of the 24 Beasley problems individually and provides LP and LGR gap percentages, nodes examined, time to best solution, time to completion, and the number of problems where optimal solutions were proven. Table 2 gives the same information for the second dataset of 640 problems and presents this information as the average results of ten problems for each problem type (D or E) and problem size. A limit of ~~3,000~~^{3,000} CPU seconds was enforced and a suboptimality tolerance of 0% was used.

Nineteen of the twenty-four Beasley problems in Table 1 were solved to optimality. In Table 2 the problems are depicted in order of increasing size. Within each problem type (D and E), problems are presented so that the number of agents is increasing and the number of tasks is decreasing; that is, such that the ratio of tasks to agents is decreasing. The results show that the algorithm performs quite well in proving optimality for problems with up to 1250 binary variables. Results for problems of size 1500 to 2000 binary variables show 57% of problems where optimality is proven. For problems of size 2250 to 2750 binary variables, 40% of the problems are solved to optimality. Twenty-nine percent of the problems with 3000 binary variables were solved to optimality. These results appear to be a step forward in increasing the size of “hard” problems that can be solved to optimality.

In the case of a large ratio of tasks to agents it is clear that both the LP and *LGR* gaps are rather tight when compared to the optimal solution (or best known solution). In fact the gaps are generally less than 0.2%. In the case of smaller ratios the LP gaps appear to be quite loose, ranging (for problems larger than 500 binary variables) from approximately 1% to 8%. However the *LGR* gaps are reasonably tight (generally less than 0.4%) although not as tight as for the larger ratio problems. In addition we hypothesize that the smaller ratio problems are more amenable to effective feasibility-based tests, effective cuts, and strong IP knapsack penalties due to relatively

1 col, .50

Table 1/ Test runs for Beasley Library Problems (Suboptimality Tolerance = 0%)

Problem type	Agents	Tasks	Tasks/ Agent	0-1 variables	Lp gap %	Lgr gap %	Total nodes	Seconds to best solution	Seconds to completion	Problems proven optimal
# A	5	100	20	# 500	0.02	0.00	# 1	# 0.1	# 0.7	# 1 of 1
B	5	100	20	500	0.63	0.27	309	4.2	6.4	1 of 1
C	5	100	20	500	0.36	0.15	295	0.1	3.7	1 of 1
# D	5	100	20	500	0.12	0.07	22504	349.9	362.2	1 of 1
# A	5	200	40	1000	0.01	0.00	1	0.1	1.0	1 of 1
B	5	200	40	1000	0.13	0.12	13409	14.7	218.8	1 of 1
C	5	200	40	1000	0.15	0.09	2152	30.4	40.6	1 of 1
# D	5	200	40	1000	0.07	0.05	114557	2937.0	3000.1	0 of 1
# A	10	100	10	1000	0.11	0.00	1	0.1	1.3	1 of 1
B	10	100	10	1000	0.45	0.15	103	0.1	2.7	1 of 1
C	10	100	10	1000	1.07	0.24	553	7.3	15.1	1 of 1
# D	10	100	10	1000	0.40	0.14	72103	2831.5	3000.1	0 of 1
# A	10	200	20	2000	0.00	0.00	1	0.2	0.8	1 of 1
B	10	200	20	2000	0.42	0.12	14137	235.9	436.7	1 of 1
C	10	200	20	2000	0.38	0.14	9550	312.6	490.4	1 of 1
# D	10	200	20	2000	0.23	0.18	48587	1896.8	3000.1	0 of 1
# A	20	100	5	2000	0.08	0.00	1	0.2	2.4	1 of 1
B	20	100	5	2000	0.93	0.12	26	0.2	6.4	1 of 1
C	20	100	5	2000	1.93	0.24	2268	90.2	115.2	1 of 1
# D	20	100	5	2000	0.93	0.41	31949	2829.4	3000.3	0 of 1
# A	20	200	10	4000	0.07	0.00	1	0.3	6.7	1 of 1
B	20	200	10	4000	0.34	0.07	775	89.0	127.2	1 of 1
C	20	200	10	4000	0.59	0.09	12179	968.7	1028.3	1 of 1
D	20	200	10	4000	0.37	0.28	15300	2375.4	3000.5	0 of 1

Rule

Note. Suboptimality tolerance = 0%. (zero)

1 col, .50

Table 2/ Test runs for Randomly Generated Problem Sets of 10 (Suboptimality Tolerance = 0%)

Problem type	Agents	Tasks	Tasks/ Agent	0-1 variables	Lp gap %	Lgr gap %	Total nodes	Seconds to best solution	Seconds to completion	Problems proven optimal
# D	5	100	20.00	# 500	# 0.14	# 0.06	# 905	# 14	# 17	# 10 of 10
# D	10	50	5.00	500	0.34	0.23	510	14	20	10 of 10
# E	5	100	20.00	500	0.16	0.06	639	17	20	10 of 10
# E	10	50	5.00	500	0.30	0.23	702	19	25	10 of 10
# D	5	150	30.00	750	0.06	0.02	2387	70	77	10 of 10
D	10	75	7.50	750	0.39	0.06	2033	60	65	10 of 10
# D	15	50	3.33	750	2.85	0.33	1113	31	48	10 of 10
# E	5	150	30.00	750	0.07	0.03	1407	42	53	10 of 10
E	10	75	7.50	750	0.50	0.09	1619	42	50	10 of 10
# E	15	50	3.33	750	1.74	0.29	12828	461	570	10 of 10
# D	5	200	40.00	1000	0.03	0.01	2759	64	68	10 of 10
D	10	100	10.00	1000	0.24	0.06	3207	108	132	10 of 10
# D	20	50	2.50	1000	5.10	0.29	467	38	48	10 of 10
# E	5	200	40.00	1000	0.04	0.03	116487	1691	2169	9 of 10
E	10	100	10.00	1000	0.34	0.15	76196	1542	3001	0 of 10
# E	20	50	2.50	1000	2.44	0.27	6013	215	346	10 of 10

Table 2 continues

Note. Suboptimality tolerance = 0%.

19

Use if table ending on two or more pp

Goes under ending rule for Tables

Table 2 continued *(Use only if necessary)*

#	D	5	250	50.00	1250	0.04	0.03	124740	2246	2696	3 of 10
	D	10	125	12.50	1250	0.24	0.12	62122	1105	3000	0 of 10
#	D	25	50	2.00	1250	3.33	0.21	2638	147	187	10 of 10
	E	5	250	50.00	1250	0.02	0.01	6098	189	204	10 of 10
	E	10	125	12.50	1250	0.14	0.03	4273	148	167	10 of 10
#	E	25	50	2.00	1250	8.29	0.44	1220	78	93	10 of 10
	D	5	300	60.00	1500	0.01	0.01	6405	332	341	10 of 10
	D	10	150	15.00	1500	0.11	0.02	7856	265	300	10 of 10
#	D	20	75	3.75	1500	1.79	0.19	7256	456	566	10 of 10
	E	5	300	60.00	1500	0.04	0.03	127116	2455	3000	0 of 10
	E	10	150	15.00	1500	0.19	0.12	54603	1895	3001	0 of 10
#	E	20	75	3.75	1500	1.23	0.30	34739	1789	3000	0 of 10
	D	10	175	17.50	1750	0.16	0.10	46037	1880	3000	0 of 10
#	D	25	70	2.80	1750	1.82	0.25	25437	1449	2399	3 of 10
	E	10	175	17.50	1750	0.08	0.02	15496	496	586	10 of 10
#	E	25	70	2.80	1750	3.62	0.23	5580	386	471	10 of 10
	D	10	200	20.00	2000	0.06	0.02	5961	312	353	10 of 10
	D	20	100	5.00	2000	0.85	0.10	17979	1360	1527	8 of 10
#	D	25	80	3.20	2000	1.41	0.32	27696	2137	3002	0 of 10
	E	10	200	20.00	2000	0.06	0.02	12929	608	746	10 of 10
	E	20	100	5.00	2000	0.85	0.37	29777	1857	3001	0 of 10
#	E	25	80	3.20	2000	2.50	0.20	11953	900	1167	10 of 10
	D	10	225	22.50	2250	0.19	0.16	38008	2098	3002	0 of 10
	D	15	150	10.00	2250	0.36	0.23	30427	1844	3001	0 of 10
#	D	25	90	3.60	2250	1.20	0.39	27512	1871	3000	0 of 10
	E	10	225	22.50	2250	0.05	0.01	11450	493	564	10 of 10
	E	15	150	10.00	2250	0.17	0.03	14338	904	1072	9 of 10
#	E	25	90	3.60	2250	1.91	0.21	16576	1332	1776	7 of 10
	D	10	250	25.00	2500	0.21	0.18	34948	2130	3002	0 of 10
	D	20	125	6.25	2500	0.64	0.36	23957	1815	3001	0 of 10
#	D	25	100	4.00	2500	1.07	0.43	24838	2093	3002	0 of 10
	E	10	250	25.00	2500	0.03	0.01	23447	1153	1222	10 of 10
	E	20	125	6.25	2500	0.41	0.05	19836	1229	1827	9 of 10
#	E	25	100	4.00	2500	1.35	0.16	20008	1642	2314	6 of 10
	D	10	275	27.50	2750	0.20	0.18	30747	1801	3000	0 of 10
#	D	25	110	4.40	2750	0.90	0.41	21986	1665	3000	0 of 10
	E	10	275	27.50	2750	0.06	0.04	29331	1748	2096	7 of 10
#	E	25	110	4.40	2750	1.06	0.14	16306	1423	2028	6 of 10
	D	10	300	30.00	3000	0.22	0.20	27723	2185	3000	0 of 10
#	D	15	200	13.33	3000	0.25	0.19	22721	1953	3000	0 of 10
	D	20	150	7.50	3000	0.49	0.32	18507	2174	3000	0 of 10
	D	25	120	4.80	3000	0.77	0.39	20301	1970	3000	0 of 10
#	D	30	100	3.33	3000	1.28	0.45	16209	2012	3001	0 of 10
	E	10	300	30.00	3000	0.03	0.01	31785	1692	2113	9 of 10
	E	15	200	13.33	3000	0.10	0.03	27500	1693	2244	5 of 10
	E	20	150	7.50	3000	0.33	0.07	22070	1956	2452	5 of 10
	E	25	120	4.80	3000	0.89	0.15	13961	1615	1921	5 of 10
	E	30	100	3.33	3000	2.29	0.27	14460	1704	2258	5 of 10

rule

Ans!

Units of measure needed? or OK?

few variables being present in any one knapsack (or assigned to any one agent).

Tables 3, 4, 5, and 6 compare our algorithm with other recently published work. Savelsbergh (1997) solved type D problems of size 10×50 and 20×50 to optimality. In Table 3 we compare his results with ours. It should be noted, however, that the results for each algorithm are for sets of ten random problems that were generated separately (but according to the same guidelines) by each author. Thus, while the problem sets are of the same size and structure, they are not the same problems. However it does appear that our algorithm outpaces the other by a substantial margin (ranging from 5 to 1 to over 65 to 1) when machine differences are taken into account.

In Table 4 we compare our algorithm with that of Park et al. (1998). They solved type D and E problems of size 5×100 and 10×50 to optimality. As with Savelsbergh, the results compare sets of problems that are different but that are generated under the same guidelines. For this limited comparison set the results show that our algorithm is comparable to PLL's for some problem sets and up to a factor of almost 40 times faster for other problem sets.

In Table 5 we compare our algorithm with that of Chu and Beasley's (CB) genetic algorithm (1997). Since CB's approach is heuristic in nature we compare the average time for CB to generate a "best solution" (see the second-to-last-column in Table 2 of their paper) with the time for our algorithm to find a solution to match or beat CB's best solution. In addition we report the time required for our algorithm to find the best solution (which may or may not be optimal) within a limit of 3000 seconds. As can be seen, on average our algorithm finds a solution equal to or better than CB's in less than one-fifth the time after adjustments are made for different machines (18.28 ~~vs.~~ ^{versus.} 96.82 seconds).

In Table 6 we compare our algorithm with the tabu-search heuristic of Laguna et al. (1995). We ran 180 problems using that heuristic for 3000 seconds for each problem. A note was made of the best solution found and the time required to find that solution. Our algorithm was run on the same problems with a limit of 3000 seconds. The solutions found in the Laguna heuristics were *not* used as an initial solution for our algorithm. As can be seen, our algorithm finds (or improves upon) the best solution found by the Laguna heuristic in a fraction of the time (20-second average versus 1747-second average). The sometimes-large gap between solutions (see the last column) can be attributed to the fact that poor solutions are sometimes returned by the Laguna heuristic.

Ans: Edits OK?

1 col .25

Table 3 Comparison of Savelsbergh and Nauss Algorithms. Suboptimality Tolerance = 0%. Problems are Randomly Generated with 10 Problems to a Set. The Comparison is for Different Sets of 10 Problems for Each Algorithm.

Problem Type	Size	Savelsbergh Algorithm		Nauss Algorithm	
		Average Nodes Examined	Maximum Nodes Examined	Average Nodes Examined	Maximum Nodes Examined
D	10 × 50	648	1896	510	1150
D	20 × 50	395	1712	467	1564
CPU Comparison		Adjusted CPU Average Seconds	Adjusted CPU Maximum Seconds	Average CPU Seconds	Maximum CPU Seconds
D	10 × 50	1377	6319	20	42
D	20 × 50	251	1172	48	137

Note. The Savelsbergh node figures and CPU times are from his paper and have been adjusted for machine differences between the Pentium II 300MHz and the IBM RS6000/590 based on the SPECFP95 benchmarks of 8.15 for the Pentium II 300MHz and 9.69 for the IBM RS6000/590. Thus, the IBM RS6000/590 is 1.19 times faster than the Pentium II 300MHz.

Table 4 Comparison of Park et al. (1998) (PLL) and Nauss Algorithms. Suboptimality Tolerance = 0%. Problems are Randomly Generated with 10 Problems to a Set. The Comparison is for Different Sets of 10 Problems for Each Algorithm.

Problem Type	Size	PLL Algorithm		Nauss Algorithm	
		Average Nodes Examined	Maximum Nodes Examined	Average Nodes Examined	Maximum Nodes Examined
D	10 × 50	10715	n/a	510	1150
E	10 × 50	611	n/a	702	1526
E	5 × 100	274	n/a	639	2013
CPU Comparison		Adjusted CPU Average Seconds	Adjusted CPU Maximum Seconds	Average CPU Seconds	Maximum CPU Seconds
D	10 × 50	779	n/a	20	42
E	10 × 50	30	n/a	25	53
E	5 × 100	23	n/a	20	50

Note. The PLL node figures and CPU times are from their paper. The CPU times have been adjusted for machine differences between the Pentium II 300MHz and the IBM RS3090/400J based on the SPECFP95 benchmarks of 8.15 for the Pentium II 300MHz and 2.76 for the IBM RS3090/400J. (This SPECFP95 is taken from PLL's paper where a SPARC station/server 10 Model 40 has a SPECFP95 of 1.38 and PLL state that the IBM3090/400J is two times faster than the SPARC 10). Thus, the Pentium II 300MHz is 2.95 times faster than the IBM RS3090/400J.

Table 5. Comparison of Chu and Beasley (1997) (CB) and Nauss Algorithms for 24 Beasley Library Problems [Suboptimality Tolerance = 0%]

Problem Type	Agents	Tasks	Tasks/Agent	0-1 Variables	Best CB Solution	Best Nauss Solution	Solution Difference Gap(%)	Nauss Execution		
								Adjusted CB Execution Seconds to Best	Seconds to Find CB Solution or Better	Nauss Execution Seconds to Nauss Best Solution
A	5	100	20	500	1698	1698	0.00	0.07	0.12	0.12
A	5	200	40	1000	3235	3235	0.00	0.04	0.06	0.06
A	10	100	10	1000	1360	1360	0.00	0.04	0.05	0.05
A	10	200	20	2000	2623	2623	0.00	2.31	0.16	0.16
A	20	100	5	2000	1158	1158	0.00	0.05	0.16	0.16
A	20	200	10	4000	2339	2339	0.00	5.89	0.28	0.28
B	5	100	20	500	1843	1843	0.00	17.22	4.18	4.18
B	5	200	40	1000	3553	3552	0.03	59.65	14.66	14.66
B	10	100	10	1000	1407	1407	0.00	4.09	0.11	0.11
B	10	200	20	2000	2831	2827	0.14	82.57	28.07	235.85
B	20	100	5	2000	1166	1166	0.00	25.99	0.17	0.17
B	20	200	10	4000	2340	2340	0.00	70.37	59.98	88.98
C	5	100	20	500	1931	1931	0.00	18.88	0.05	0.05
C	5	200	40	1000	3458	3456	0.06	72.09	29.16	30.43
C	10	100	10	1000	1403	1402	0.07	23.15	5.60	7.25
C	10	200	20	2000	2814	2806	0.29	85.31	41.25	312.64
C	20	100	5	2000	1244	1243	0.08	37.99	54.76	90.19
C	20	200	10	4000	2397	2391	0.25	148.73	71.29	968.66
D	5	100	20	500	6373	6353	0.31	50.20	6.27	349.93
D	5	200	40	1000	12796	12745	0.40	226.09	17.08	2937.03
D	10	100	10	1000	6379	6349	0.47	118.10	9.67	2831.47
D	10	200	20	2000	12601	12447	1.24	375.75	22.41	1896.80
D	20	100	5	2000	6269	6200	1.11	236.97	16.75	2829.38
D	20	200	10	4000	12452	12263	1.54	662.07	56.52	2375.42
Averages								96.82	18.28	623.92

Note: CB execution times are from their paper and have been adjusted for SPECFP95 machine differences between the Pentium II 300Mhz (8.15) and the Silicon Graphics Indigo RS5000 (2.8). A comparison of the SGI RS4000 and the RS5000 was based on the SPECFP92 benchmarks of 50 for the RS5000 and 19 for the RS4000. Thus, an equivalent SPECFP95 benchmark for the SGI RS 4000 is 1.11, so the Pentium II 300Mhz is 7.37 times faster than the SGI RS4000.

Suboptimality tolerance = 0% 23

1 col., .50

1 col., .50

#

Table 6. Comparison of Laguna et al. (1995) Heuristic and Nauss Algorithm on Test Runs for Randomly Generated Problem Sets of 10 Suboptimality Tolerance for Nauss Algorithm = 0.1%. Laguna et al. (1995) Heuristic and Nauss Algorithm were Both Run for 3000 Seconds Each on a Pentium II 300MHz Machine.

Problem Type	Agents	Tasks	Tasks/ Agent	Average Laguna		Average Nauss		Average Nauss		% Gap Between Laguna Solution and Nauss Solution
				Average Solution	Seconds to Best Solution	Average Solution	Seconds to Find Laguna Solution or Better			
D	5	200	40.00	26214	2076	23442	11	11.82		
D	10	100	10.00	11085	1589	11059	10	0.24		
D	20	50	2.50	2605	1772	2597	10	0.31		
E	5	200	40.00	12977	1423	12971	26	0.05		
E	10	100	10.00	6441	1301	6432	7	0.14		
E	20	50	2.50	3244	1505	3234	7	0.31		
D	10	200	20.00	24491	2859	22447	20	9.11		
D	20	100	5.00	8639	1572	8579	22	0.70		
D	25	80	3.20	5127	747	5094	13	0.65		
E	10	200	20.00	23842	2872	22100	18	7.88		
E	20	100	5.00	6355	1457	6319	14	0.57		
E	25	80	3.20	4735	983	4684	18	1.09		
D	10	300	30.00	19063	2172	19036	42	0.14		
D	20	150	7.50	9495	1415	9454	26	0.43		
D	30	100	3.33	6417	1286	6367	21	0.79		
E	10	300	30.00	75442	2867	33610	37	124.46		
E	20	150	7.50	15917	2018	15719	32	1.26		
E	30	100	3.33	5772	1536	5688	33	1.48		
Averages					1747		20	8.97		

Note. For the 180 test problems, the "best Nauss solution" was strictly better than the "best Laguna solution" in 175 cases. In the remaining 5 problems the "best Nauss solution" value and the "best Laguna solution" value were the same. For all 180 test problems, the time required for the Nauss algorithm to match or beat the "best Laguna solution" was smaller than the Laguna time.

Table 7 compares our algorithm with CPLEX 6.6 on 40 smaller problems ranging from 500 to 1000 binary variables. Both algorithms were run on a Pentium II Xeon 450 MHz workstation with 448 MB of ECC RAM. As can be seen, our algorithm solves these problems from 2 to 5.5 times faster. For ten of the problems that could not be solved by our algorithm or CPLEX, the average best solution found was 0.28% better using our algorithm.

7. Conclusions

While our computational results look favorable, it appears that improvements may still be possible both in terms of reducing solution time for given problems and in terms of solving even larger problems. Since 30% to 60% of CPU time is devoted to solving knapsack problems, it should be possible to achieve appreciable time savings in two ways. First, a more efficient knapsack code than the one used in this paper would almost assuredly reduce times, but the field of candidates is reduced if the r_{ij} are allowed to be real-valued (Martello and Toth 1990). However one needs to exercise care in making time-saving approximations since many of the knapsacks are relatively small and there is a setup cost that is incurred regardless of knapsack size. Second, a parallel-processing design could be used to solve either the m independent knapsacks in the Lagrangean relaxation phase or could be used to solve multiple nodes of the tree simultaneously. It is also possible that the Laguna heuristic might be used at various places in the branch-and-bound tree where a Lagrangean relaxation solution might be used as an initial point in the tabu search. For example, the Laguna heuristic could be used in “hot start” mode in step 7 of the algorithm. Certainly another avenue to explore would be the incorporation of additional cuts as developed by Gottlieb and Rao (1990). A tightened LP relaxation might lead to a tighter Lagrangean relaxation. Finally improved penalties utilizing the property of separability of the multiple choice constraints as well as the knapsack constraints (extending Section 3.3) might result in reducing the size of the branch-and-bound tree.

Finally, we state that the algorithm at hand, while able to solve large, hard problems to optimality, may also be used in a “heuristic” mode to generate good solutions more quickly than many known heuristic approaches. Such results should instill renewed vigor within the optimization community to continue to meld optimization algorithms with heuristic approaches.

Table 7 Comparison of Nauss Algorithm and CPLEX 6.6 Suboptimality Tolerance = 0.01% (CPLEX Default)

Problem Type	Size	Nauss Algorithm		CPLEX 6.6	
		Average Nodes Examined	Maximum Nodes Examined	Average Nodes Examined	Maximum Nodes Examined
D	5 x 100	628	946	1512	4570
E	5 x 100	727	2278	1089	3131
D	10 x 100	4634	22760	23605	56516
E	10 x 100	51935	59129	87563	117319
CPU Comparison		Average CPU Seconds	Maximum CPU Seconds	Average CPU Seconds	Maximum CPU Seconds
D	5 x 100	5.93	7.96	12.78	34.55
E	5 x 100	6.59	14.67	10.70	27.75
D	10 x 100	81.19	358.43	451.49	1200.03
E	10 x 100	1200.08	1200.28	1200.03	1200.03
Optimal Solution Found		Number of Problems		Number of Problems	
D	5 x 100	10		10	
E	5 x 100	10		10	
D	10 x 100	10		8	
E	10 x 100	0		0	
Optimal (or Best Feasible) Gap Between Nauss and CPLEX (Gap = 100(CPLEX - NAUSS)/NAUSS)		Average % Gap		Maximum % Gap	
D	5 x 100	0.00		0.00	
E	5 x 100	0.00		0.00	
D	10 x 100	0.01		0.05	
E	10 x 100	0.28		0.55	

Note: Problems are randomly generated with 10 problems to a set and run on a 450MHz Pentium II Xeon with 448MB of ECC RAM for a maximum of 1200 CPU seconds run time. Nauss algorithm run without Laguna heuristic.

Acknowledgments 35

The author ~~would like to~~ thank (Manuel Laguna for supplying a copy of his code for use in the computational experiments, John Beasley for making available some of the test problems, and the anonymous referees for suggestions on improving the paper.

References

- Amini, M., M. Racer. 1994. A rigorous computational comparison of alternative solution methodologies for the generalized assignment problem. *Management Science* 40 868-890.
- Balachandran, V. 1972. An integer generalized transportation model for optimal job assignment in computer networks. Working Paper 34-72-3, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA.
- Balas, E., R. Jeroslow. 1972. Canonical cuts in the unit hypercube. *SIAM J. Appl. Math.* 23 61-69.
- Barr, R., B. Golden, J. Kelly, M. Resende, W. Stewart. 1995. Designing and reporting on computational experiments with heuristic methods. *Journal of Heuristics* 1 9-32.
- Cattysse, D., Z. Degraeve, J. Tistaert. 1998. Solving the generalized assignment problem using polyhedral results. *European J. Operational Research* 108 618-628.
- Cattysse, D., M. Salomon, L. Van Wassenhove. 1994. A set partitioning heuristic for the generalized assignment problem. *European J. Operational Research* 72 167-174.
- Cattysse, D., L. Van Wassenhove. 1992. A survey of algorithms for the generalized assignment problem. *European J. Operational Research* 60 260-272.
- Chu, P.C., J.B. Beasley. 1997. A genetic algorithm for the generalized assignment problem. *Computers and Operations Research* 24 17-23.
- Fisher, M., R. Jaikumar. 1981. A generalized assignment heuristic for vehicle routing. *Networks* 11 109-124.
- Gavish, B., H. Pirkul. 1991. Algorithms for the multi-resource generalized assignment problem. *Management Science* 37 695-713.
- Geoffrion, A.M. 1974. Lagrangean relaxation for integer programming. *Mathematical Programming Study* 1: Approaches to Integer Programming 82-114.
- Geoffrion, A.M., R.E. Marsten. 1972. Integer programming algorithms: a framework and state-of-the-art survey. *Management Science* 18 465-491.
- Gottlieb, E., M. Rao. 1990. The generalized assignment problem: valid inequalities and facets. *Mathematical Programming* 46 31-52.

Ans. to
Apps. to
Int.
Progr.
Vol. 2?
Edit
OK?

- Guignard, M., M. Rosenwein. 1989. An improved dual based algorithm for the generalized assignment problem. *Operations Research* 37 658-663.
- Guignard, M., S. Zhu. 1996. A two phase dual algorithm for solving lagrangean duals in mixed integer programming. Working Paper, The Wharton School, University of Pennsylvania, Philadelphia, PA.
- Laguna, M., J. Kelly, J. Gonzalez-Velarde, F. Glover. 1995. Tabu search for the multilevel generalized assignment problem. *European J. Operational Research* 82 176-189.
- Marsten, R. 1980. The design of the XMP linear programming library. Technical Report, 80-2, Department of Management Information Systems, University of Arizona, Tucson, AZ.
- Martello, S., P. Toth. 1981. An algorithm for the generalized assignment problem. J.P. Brans, ed. *Operational Research* 81. North-Holland, Amsterdam, The Netherlands. 589-603.
- Martello, S., P. Toth. 1990. *Knapsack Problems, Algorithms, and Computer Implementations*. John Wiley and Sons, New York.
- Mazzola, J., A. Neebe, C. Dunn. 1989. Production planning of a flexible manufacturing system in a material requirements planning environment. *International Journal of Flexible Manufacturing Systems* 1/2 115-142.
- Nauss, R.M. 1976. An efficient algorithm for the 0-1 knapsack problem. *Management Science* 23 27-31.
- Park, J.S., B.H. Lim, Y. Lee. 1998. A lagrangian dual-based branch-and-bound algorithm for the generalized multi-assignment problem. *Management Science* 44 271-282.
- Ronen, D. 1992. Allocation of trips to trucks operating from a single terminal. *Computers and Operations Research* 19 129-138.
- Ross, G.T., R.M. Soland. 1975. A branch and bound algorithm for the generalized assignment problem. *Mathematical Programming* 8 91-103.
- Ross, G.T., R.M. Soland. 1977. Modeling facility location problems as generalized assignment problems. *Management Science* 24 345-357.
- Savelsbergh, M. 1997. A branch-and-price algorithm for the generalized assignment problem. *Operations Research* 45 831-841.
- Watcom FORTRAN 77 User's Guide. 1995. Watcom International Corp., Waterloo, Ontario, Canada.

Accepted by Richard S. Barr; received March 1999; revised June 2000, May 2002; accepted January 2003.