CIRRELT

Centre interuniversitaire de recherche
sur les réseaux d'entreprise, la logistique et le transport

Interuniversity Research Centre
on Enterprise Networks, Logistics and Transportation

An Exact Method with Variable Fixing for Solving the Generalized Assignment Problem

Marius Posta
Jacques A. Ferland
Philippe Michelon

March 2011

CIRRELT-2011-17

Bureaux de Montréal :
Université de Montréal
C.P. 6128, succ. Centre-ville
Montréal (Québec)
Canada H3C 3J7
Téléphone : 514 343-7575
Télécopie : 514 343-7121

Bureaux de Québec :
Université Laval
2325, de la Terrasse, bureau 2642
Québec (Québec)
Canada G1V 0A6
Téléphone : 418 656-2073
Télécopie : 418 656-2624

www.cirrelt.ca

An Exact Method with Variable Fixing for Solving the Generalized Assignment Problem

Marius Posta[1,2,3,*], Jacques A. Ferland[1,2], Philippe Michelon[3]

[1] Interuniversity Research Centre on Enterprise Networks, Logistics and Transportation (CIRRELT)

[2] Department of Computer Science and Operations Research, Université de Montréal, C.P. 6128, succursale Centre-ville, Montréal, Canada H3C 3J7

[3] Université d'Avignon et des Pays du Vaucluse, Laboratoire d'informatique d'Avignon, F-84911 Avignon, Cedex 9, France

Abstract. We propose a simple and very effective algorithm for solving the generalized assignment problem exactly. Our contribution is twofold: we reformulate the optimization problem into a sequence of decision problems, and we solve these effectively using variable-fixing rules. The decision problems are solved by a simple depth-first lagrangian branch-and-bound method, improved by the variable-fixing rules which help prune the search tree. These rules rely on lagrangian relative costs which we compute using an existing but little-known dynamic programming algorithm.

Keywords. Integer programming, generalized assignment problem, branch and bound, Lagrangian relaxation, dynamic programming.

---

* Corresponding author: Marius.Posta@cirrelt.ca

Dépôt légal – Bibliothèque et Archives nationales du Québec
Bibliothèque et Archives Canada, 2011

1 Introduction

The Generalized Assignment Problem (GAP), originally specified by Ross and Soland [11], consists in assigning jobs to agents subject to resource constraints. Specifically, each job $j \in \{1, \ldots, n\}$ must be assigned to one and only one agent $i \in \{1, \ldots, m\}$. Assigning job j to agent i incurs a cost c_{ij} and consumes a_{ij} resources of that agent having a resource capacity b_i. These coefficients are all assumed to be integers. The mathematical model for (GAP) is as follows:

$$\begin{array}{lll} \min & \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij} & \\ \text{s.t.} & & \\ & \sum_{j=1}^{n} a_{ij} x_{ij} \leq b_i, & i = 1, \ldots, m, \quad (1) \\ & \sum_{i=1}^{m} x_{ij} = 1, & j = 1, \ldots, n, \quad (2) \\ & x_{ij} \in \{0, 1\}, & i = 1, \ldots, m, \\ & & j = 1, \ldots, n. \end{array}$$

Several exact search methods presented in the literature are branch-and-bound methods using the lagrangian relaxation obtained by dualizing the assignment constraint (2). If λ denotes the vector of multipliers, then this relaxation decomposes the problem into m 0-1 knapsack problems:

$$\begin{array}{lll} z(\lambda) = \sum_{j=1}^{n} \lambda_j + \min & \sum_{i=1}^{m} (\sum_{j=1}^{n} (c_{ij} - \lambda_j) x_{ij}) & \\ \text{s.t.} & & \\ & \sum_{j=1}^{n} a_{ij} x_{ij} \leq b_i, & i = 1, \ldots, m, \\ & x_{ij} \in \{0, 1\}, & i = 1, \ldots, m; j = 1, \ldots, n. \end{array}$$

The langrangian dual bound $\max_\lambda z(\lambda)$ thus obtained is at least as strong as the linear relaxation bound, which is equal to the langrangian dual bounds obtained by dualizing either the knapsack constraints, or both knapsack and assignment constraints, as shown in [13].

Both Haddadi and Ouzia [6] and Nauss [8] use this lagrangian dualizing the assignment constraints in their branch-and-bound methods. The contribution of Haddadi and Ouzia [6] is a procedure for repairing the solutions of the lagrangian relaxation obtained during the subgradient search into feasible solutions which might improve the upper bound. Nauss [8] presents a hybrid linear and langrangian-relaxation-based method featuring knapsack cover cut generation. Savelsbergh [12] introduces a branch-and-price method, in which the columns generated correspond to feasible assignments for an agent. This method has been improved by Pigatti et al. [9] who propose a scheme for stabilizing the multipliers during the column-generation phase in order to improve convergence. More recently Avella et al. [2] introduced the best exact search method in the literature, their branch-and-cut method uses the linear relaxation supplemented by a sophisticated cut generation scheme. Furthermore, several heuristic methods have been proposed, in particular Yagiura et al. introduce a metaheuristic method based on an ejection chain neighborhood [14] [15] and Diaz and Fernandez [4] propose a simple and effective tabu search method.

In this paper we introduce a method where the optimization problem is replaced by a sequence of decision problems. A valid lower bound is computed for (GAP) and then a search is performed for a feasible solution with the value of the lower bound. If none is found, the lower bound is incremented and the procedure is repeated. This continues until a feasible solution is found, which is then optimal. Naturally, our algorithm can be efficient only if the gap between the initial lower bound and the optimal solution is small.

Section 2 includes an overview of our algorithm. The variable-fixing procedure used by our algorithm is described in section 3. This procedure relies heavily on the notion of relative costs of the assignment variables, which are obtained by a dynamic programming approach based on results due to Karabakal et al. [7] summarized in section 4. Section 5 presents the numerical results obtained by running our implementation on the Beasly instance set [3]. The computation time is often much less than that required by Avella et al. [2]. Furthermore, we provide the previously unknown optimal values for three instances.

2 Solving a sequence of decision problems

Our approach can be outlined as follows:

- Denote by $\bar{z}$ a lower bound on the optimal value of the problem.
- Solve the following decision problem $GAP(\bar{z})$:
  - Does a feasible solution of cost $\bar{z}$ or less exist?
- If the answer is yes, then the solution is optimal, otherwise increment $\bar{z}$ and repeat.

This very crude algorithm relies on the fact that solution values are integer, and thus transforms the optimization problem into solving a sequence of decision problems. We use the lagrangian relaxation dualizing the assignment constraints (2) outlined in the previous section to compute the initial lower bound, and also to compute the lower bounds for the nodes in the branch-and-bound method to solve the decision problems $GAP(\bar{z})$. Our branch-and-bound method to solve $GAP(\bar{z})$ can be outlined as follows:

1. Initialize the branch-and-bound active node list with the root node.
2. If the active node list is empty, then the answer to the decision problem is NO. Otherwise, select and remove a node from the list.
3. Compute a lower bound for this node by solving the corresponding lagrangian dual. If, during the optimization of the dual, we obtain a relaxed solution which is feasible and verifying $z(\lambda) \leq \bar{z}$, then the answer to the decision problem is YES, and the search is over.

4. If the lower bound exceeds the upper bound $\bar{z}$, go back to step 2.
5. Call the variable-fixing procedure.
6. Select the job j with the highest multiplier λ_j such that not all x_{ij} are fixed, and branch on each unfixed agent, thus creating up to m child nodes. Add the child nodes to the active node list and go back to step 2.

Perhaps surprisingly, the numerical results in section 5 indicate that this approach works very well in practice. We can offer a few tentative explanations on why this is the case. Firstly, the bound provided by $\max_\lambda z(\lambda)$ is usually good enough that the sequence of decision problems to be solved is very short, thus limiting the replication of computational effort from solving very similar decision problems. As we pointed out previously, finding good feasible solutions for (GAP) can be challenging and as a consequence traditional branch-and-bound methods can perform very poorly.

Secondly, the variable-fixing procedure in step 5 is a key component of our algorithm since it significantly speeds up the resolution of the decision problems. Indeed it uses logical inference to fix variables to their locally optimal value based on the global upper bound $\bar{z}$ as well as information provided by the lagrangian relaxation. Obviously, a proper implementation of this procedure is critical to speed up the resolution of the successive decision problems. It is especially effective when the optimality gap is tight, and our decision-problem-sequence approach exploits this well. In the following sections, we present this procedure in detail.

Finally, current branch-and-bound methods and even the best metaheuristics often fail to identify good feasible solutions but we manage to bypass this problem entirely with our approach. Indeed, before developing the method presented in this paper, we tried to solve the optimization problem using a traditional branch-and-bound method (essentially that proposed by Haddadi and Ouzia [6]) enhanced with our variable-fixing rules. However, this was not improving the performance in all cases. Indeed, the variable-fixing rules were useless before reaching a good enough feasible solution, and for some instances finding a feasible solution is difficult. The method presented in this paper performs with a more consistent behavior. Additional details are included in our concluding analysis in subsection 5.4.

3 Variable fixing

As pointed out by Atamtürk and Savelsbergh in their survey [1], variable-fixing procedures are used in many linear-relaxation-based solvers. Our procedure is based on similar principles, but they are adapted to our lagrangian relaxation for (GAP). Consider any node of the branch-and-bound tree, denote by:

- λ^* the best multipliers found during the search for $\max_\lambda z(\lambda)$[1],

[1] Thus the value $z(\lambda^*)$ is a lower bound for this node

- Δ the local optimality gap, i.e. $\Delta = \bar{z} - z(\lambda^*)$,
- $x(\lambda^*) \in \{0,1\}^{mn}$ the optimal value of the lagrangian relaxation using the multipliers λ^*.

Recall that the solution to the lagrangian relaxation is a binary vector. At any node, some components of this vector have been fixed to either 0 or 1, either through earlier branching or through earlier variable fixing. For any unfixed component x_{ij}, if we fix x_{ij} to its value $x(\lambda^*)_{ij}$, then $z(\lambda^*)$ does not change. However, if we fix x_{ij} to its complementary value $(1 - x(\lambda^*)_{ij})$, then $z(\lambda^*)$ may increase. If it increases above $\bar{z}$ we can conclude that no optimal solution can be found with $x_{ij} = 1 - x(\lambda^*)_{ij}$, hence we may safely fix x_{ij} to the value $x(\lambda^*)_{ij}$.

Of course, computing $z(\lambda^*)$ anew after fixing each unfixed component x_{ij} may be unreasonable from a computational point of view. Suppose however we have $c(\lambda^*) \in \mathbb{R}^{mn}$, a vector of relative costs associated to $x(\lambda^*)$. A relative cost $c(\lambda^*)_{ij}$ associated to the variable $x(\lambda^*)_{ij}$ is a lower bound on the increase of the objective value of the relaxation when x_{ij} is forced to its complementary value $(1 - x(\lambda^*)_{ij})$. As a consequence, considering any unfixed variable x_{ij}:

- if $x(\lambda^*)_{ij} = 0$ and $c(\lambda^*)_{ij} > \Delta$, then x_{ij} can be fixed to 0;
- if $x(\lambda^*)_{ij} = 1$ and $c(\lambda^*)_{ij} > \Delta$, then x_{ij} can be fixed to 1 and x_{kj} to 0 for all $k \neq i$.

The preceding rules can be seen as 'simple' variable fixation rules, because we may use more powerful rules by exploiting the structure of the problem, based on the fact that each job has to be assigned to one and only one agent, and the fact that the lagrangian relaxation decomposes into m independent knapsack problems. Consider any unfixed variable x_{ij}:

- Suppose that $x(\lambda^*)_{ij} = 0$ and $\sum_{k=1}^{m} x(\lambda^*)_{kj} \geq 1$:
  if x_{ij} were to take value 1 in a feasible solution, then all other x_{kj} would have to take value 0, therefore if

$$c(\lambda^*)_{ij} + \sum_{\substack{k=1 \\ x(\lambda^*)_{kj}=1}}^{m} c(\lambda^*)_{kj} > \Delta,$$

  then x_{ij} can be fixed to 0.

- Suppose that $x(\lambda^*)_{ij} = 1$ and $x(\lambda^*)_{kj} = 0$ for all $k \neq i$:
  if x_{ij} were to take value 0 in a feasible solution, then some x_{kj}, $k \neq i$, not fixed to 0 would have to take value 1, therefore if

$$c(\lambda^*)_{ij} + \min\{c(\lambda^*)_{kj} \mid x_{kj} \text{ unfixed}, k \in \{1, \ldots, i-1, i+1, \ldots, m\}\} > \Delta,$$

  then x_{ij} can be fixed to 1 and x_{kj} can be fixed to zero for all $k \neq i$.

- Suppose that x_{kj} is fixed to 0 for all $k \neq i$:
  obviously, x_{ij} can be fixed to 1. However if $x(\lambda^*)_{ij} = 0$ and $c(\lambda^*)_{ij} > \Delta$, then the current node can be culled.

Indeed we may sometimes be able to strengthen the lower bound enough to be able to cull the current node. Considering now any job j:

- Suppose that $\sum_{i=1}^{m} x(\lambda^*)_{ij} = 0$:
  at least one variable x_{ij} for some $i \in \{1, \ldots, m\}$ must take value 1. Therefore if
  $$\min \{c(\lambda^*)_{ij} \mid x_{ij} \text{ unfixed}, i \in \{1, \ldots, m\}\} > \Delta,$$
  then the current node can be culled.

- Suppose that x_{ij} is fixed to 0 for all $i \in \{1, \ldots, m\}$:
  obviously, the current node can be culled.

Maybe it is possible to derive yet more rules, however these work well enough: only considering rules which fix variables to their current value in $x(\lambda^*)$ allows us to apply them successively to all unfixed variables without needing to recompute $c(\lambda^*)$ and $x(\lambda^*)$ each time. In the following section, we describe how we compute the relative costs $c(\lambda^*)$, upon which this variable fixation scheme depends.

4 Lagrangian relative costs

The algorithm to determine the relative costs is based on that introduced by Karabakal et al in [7]. However, in [7] the output of this algorithm was used within a steepest-descent heuristic for solving the lagrangian dual $\max_\lambda z(\lambda)$, and not for determining relative costs for fixing variables in the branch-and-bound. Indeed, to our knowledge, we are the first to do so.

We now summarize how Karabakal et al. obtain the relative costs $c(\lambda)$ for any multiplier and at any node in the branch-and-bound search. However, for simplicity and without loss of generality, we shall assume that we are at the root node, and that none of the variables x_{ij} have been fixed yet.

4.1 Decomposition of the lagrangian relaxation

Recall that when solving the lagrangian relaxation for any vector of multipliers λ, we can decompose the problem into m independent knapsack subproblems. Denote by $z^i(\lambda)$ the optimal value of the knapsack subproblem corresponding to agent i given multipliers λ:

$$z(\lambda) = \sum_{j=1}^{n} \lambda_j + \sum_{i=1}^{m} z^i(\lambda)$$

and

$$\begin{aligned} z^i(\lambda) = \min \quad & \textstyle\sum_{j=1}^{n}(c_{ij} - \lambda_j)x_{ij} \\ \text{s.t.} \quad & \\ & \textstyle\sum_{j=1}^{n} a_{ij}x_{ij} \leq b_i, \\ & x_{ij} \in \{0,1\}, \qquad j = 1,\ldots,n. \end{aligned}$$

Denote by:

- $x(\lambda)$ the optimal solution to the lagrangian relaxation with multipliers λ,
- $z^i(\lambda, x_{ij} = 0)$ the optimal value of the knapsack subproblem when a free variable x_{ij} is fixed to 0, and
- similarly $z^i(\lambda, x_{ij} = 1)$ when x_{ij} is fixed to 1.

Fixing a free variable x_{ij} to its value in $x(\lambda)$ will not change $z(\lambda)$, since by definition $x(\lambda)$ is the optimal solution to the lagrangian relaxation given λ. However fixing x_{ij} to its complement $(1 - x(\lambda)_{ij})$ may increase the objective value of the knapsack subproblem i from $z^i(\lambda)$ to $z^i(\lambda, x_{ij} = 1 - x(\lambda)_{ij})$. We may therefore define the vector of lagrangian relative costs $c(\lambda)$ as the vector of these modifications:

$$c(\lambda)_{ij} = \begin{cases} z^i(\lambda, x_{ij} = 0) - z^i(\lambda) & \text{if } x(\lambda)_{ij} = 1, \\ z^i(\lambda, x_{ij} = 1) - z^i(\lambda) & \text{if } x(\lambda)_{ij} = 0, \end{cases}$$

At this stage we already know $z^i(\lambda)$ for all i, therefore computing $c(\lambda)$ requires computing $z^i(\lambda, x_{ij} = 1 - x(\lambda)_{ij})$ for all (i, j). We now show how to do this efficiently using a dynamic programming approach.

4.2 Dynamic programming approach

Since the values a_{ij} are all integer, we can solve the knapsack subproblem associated with each agent $i \in \{1,\ldots,m\}$ using a dynamic programming approach. For each agent i, we define an initial state s^i, a final state t^i, and the states $v_\beta^{i,j}$ associated with each job $j \in \{1,\ldots,n\}$ and each possible resource usage $\beta \in \{0,\ldots,b_i\}$.

We then define the following transition mechanism, given a vector of multipliers λ. Consider any state $v_\beta^{i,j}$ with $1 \leq j < n$ and $0 \leq \beta \leq b_i$:

- we allow transition to state $v_\beta^{i,j+1}$ at zero cost,
- we allow transition to state $v_{\beta+a_{ij}}^{i,j+1}$ at cost $c_{ij} - \lambda_j$ if $\beta + a_{ij} \leq b_i$.

We also allow transition from s^i to $v_\beta^{i,1}$ as well as from $v_\beta^{i,n}$ to t^i for all $\beta \in \{0,\ldots,b_i\}$ at zero cost. The state space for the knapsack problem for agent i parameterized by the multipliers λ can be represented as a weighted directed acyclic graph $G^i(\lambda)$ illustrated in figure 1.

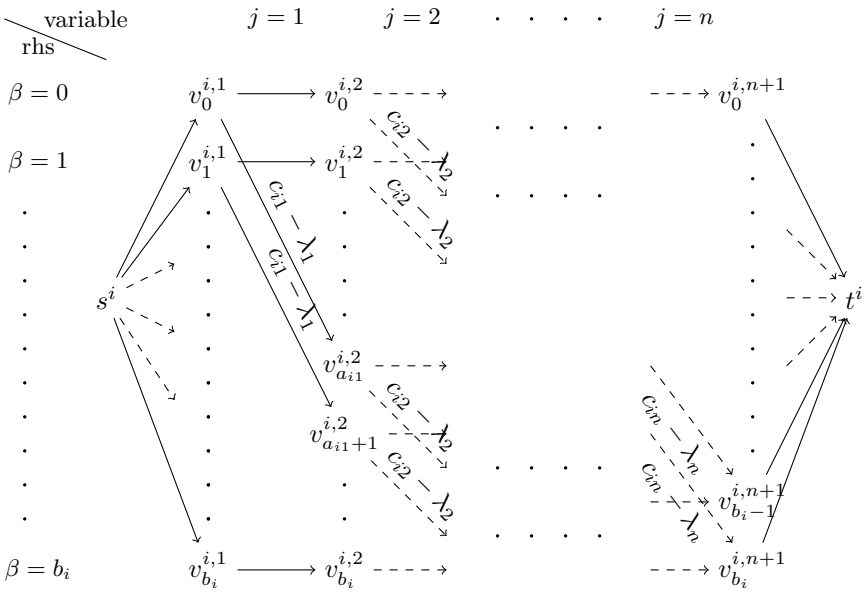

Figure 1: The graph $G^i(\lambda)$.

By representing the knapsack subproblem as a shortest-path problem, we see that the optimal value $z^i(\lambda)$ is the length of the shortest path from s^i to t^i in $G^i(\lambda)$. Denote by $f^i(\lambda, j, \beta)$ the length of the shortest path from s^i to $v_\beta^{i,j}$, and by $g^i(\lambda, j, \beta)$ the length of the shortest path from $v_\beta^{i,j}$ to t^i. Following the Bellman optimality principle, these values can be expressed recursively:

$$f^i(\lambda, j, \beta) = \begin{cases} 0 & \text{if } j = 1, \\ f^i(\lambda, j-1, \beta) & \text{if } \beta < a_{ij}, \\ \min \left\{ \begin{array}{c} f^i(\lambda, j-1, \beta), \\ f^i(\lambda, j-1, \beta - a_{ij}) + c_{ij} - \lambda_j \end{array} \right\} & \text{otherwise,} \end{cases}$$

$$g^i(\lambda, j, \beta) = \begin{cases} 0 & \text{if } j = n+1, \\ g^i(\lambda, j+1, \beta) & \text{if } \beta > b_i - a_{ij}, \\ \min \left\{ \begin{array}{c} g^i(\lambda, j+1, \beta), \\ g^i(\lambda, j+1, \beta + a_{ij}) + c_{ij} - \lambda_j \end{array} \right\} & \text{otherwise.} \end{cases}$$

Notice that we may obtain $z^i(\lambda)$ by computing either $f^i(\lambda, n, b_i)$ or $g^i(\lambda, 0, 0)$ using a dynamic programming approach.

We now show how this approach also leads itself to solving the knapsack problem for agent i given the multipliers λ in which a variable x_{ij} has been fixed to a specific value, i.e. $z^i(\lambda, x_{ij} = 0)$ or $z^i(\lambda, x_{ij} = 1)$. Fixing a variable x_{ij} to 1 consists in removing a subset of arcs from the state space $G^i(\lambda)$, namely the arcs $(v_\beta^{i,j}, v_\beta^{j+1})$ for all $\beta \in \{0, \ldots, b_i\}$. Similarly, fixing a variable x_{ij} to 0 consists in removing from $G^i(\lambda)$ all arcs $(v_\beta^{i,j}, v_{\beta+a_{ij}}^{i,j+1})$ with $\beta \in \{0, \ldots, b_i\}$. Fortunately,

the graph $G^i(\lambda)$ is topologically ordered, and it is relatively easy to compute the length of the new shortest path. Indeed, according to the Bellman optimality principle, we have

$$z^i(\lambda, x_{ij} = 0) = \min_{0 \leq \beta \leq b_i} \left(f^i(\lambda, j, \beta) + g^i(\lambda, j+1, \beta) \right).$$

and similarly,

$$z^i(\lambda, x_{ij} = 1) = c_{ij} - \lambda_j + \min_{0 \leq \beta \leq b_i - a_{ij}} \left(f^i(\lambda, j, \beta) + g^i(\lambda, j+1, \beta + a_{ij}) \right).$$

As a consequence the relative costs $c(\lambda)$ can be found in $O(mnb_i)$.

The case where a variable $x_{ij'}$ is fixed to 1 is illustrated in figure 2. The ars $(v_\beta^{i,j'}, v_\beta^{i,j'+1})$ are removed from $G^i(\lambda)$ for all $\beta \in \{0, \ldots, b_i\}$. If we assume that the shortest path marked in full thick lines corresponds to the value of $z^i(\lambda)$, then to obtain $z^i(\lambda, x_{ij'} = 1)$ we need to examine each arc $(v_\beta^{i,j'}, v_{\beta + a_{ij'}}^{i,j'+1})$ for all $\beta \in \{0, \ldots, b_i - a_{ij'}\}$ to determine the new shortest path (marked in dashed thick lines).

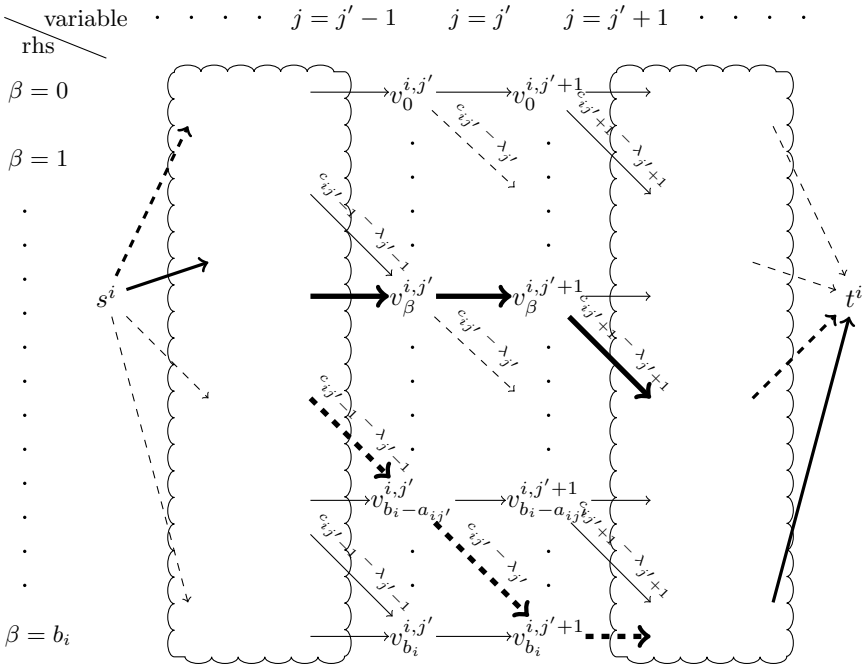

Figure 2: The new shortest path after fixing $x_{ij'}$ to 1.

5 Computational experience

First we introduce a few details pertaining to the implementation of our method which warrant further discussion. We then specify the experimental protocol applied for obtaining our numerical results. These are then presented and compared with the best known results. Finally we conclude by analyzing the efficiency of our variable fixing procedure.

5.1 Implementation details

The efficient resolution of knapsack problems in the implementation of our method is a matter of great importance. These knapsack problems result from the decomposition of the lagrangian relaxation of (GAP), and in our method this relaxation is used in two places: in the resolution of the lagrangian dual, and in the computation of lagrangian relative costs. In the latter case we use dynamic programming as explained previously in subsection 4.2, however in the former we use a more sophisticated algorithm, MINKNAP, developed by David Pisinger [10]. This choice proved essential to solving the difficult problem instances in which the coefficients are strongly correlated. Interestingly, Avella et al. [2] who obtained the best known results also use MINKNAP for their knapsack cover cut generation. Until now in the Generalized Assignment Problem literature, such knapsack problems were solved either using a simple branch-and-bound or even using dynamic programming.

Also until now, it seems that previous authors using lagrangian relaxations solved the lagrangian dual with a subgradient method. We choose instead to solve it with a bundle method, and more specifically specifically using Antonio Frangioni's [B]TT/OBP solver [5]. The bundle method is robust, requiring fewer parameter adjustments than the the subgradient method to perform acceptably. In fact we found that allowing up to around a hundred bundle iterations worked well enough for all problems in our instance set, and did not tweak the method further. On the contrary, despite all our efforts in tweaking the magic numbers for the subgradient method, we never came close to performing as well as the bundle method. Another motivation for preferring the bundle method is that in solving $\max_\lambda z(\lambda)$, the computational bottleneck is clearly in solving $z(\lambda)$ for a given λ, rather than in updating and optimizing the bundle model of $\{z(\lambda) \mid \lambda \in \mathbb{R}^n\}$.

When solving a decision problem $GAP(\bar{z})$, we naturally choose to use a depth-first node selection strategy for the branch-and-bound search, since we wish to find a feasible solution as soon as possible. If there are none, the node selection strategy has no impact other than on memory use, which fortunately is minimized when going depth-first. Also, when evaluating the lower bound in all nodes (except at the root node), the bundle method is initialized with the best multipliers found for the parent node, and is run until a fixed iteration limit is reached (around 100 works well), or until the bundle method performs a minor step of size below a certain threshold (10^-5).

For the root node, we re-use the multipliers obtained during the computation of the initial lower bound (which also yields the initial $\bar{z}$), thus avoiding some replication of effort. Note that it is possible to extend this approach to other nodes of the search tree where no variables have been fixed other than by branching, since we find these nodes in the search tree for the next decision problem $GAP(\bar{z}+1)$. We have not bothered to do this, after we noticed in our computations that the variable-fixing procedure is effective very early on in the search, in most cases fixing variables in all nodes of a search tree.

We compute the initial lower bound as follows. First we solve the linear relaxation of the MIP model of the instance with CPLEX. We then initialize the bundle method with the optimal knapsack constraint multipliers of the linear relaxation, and we let it run with a much higher iteration limit (around 100,000).

5.2 Experimental protocol

The best known results for these problems are provided by Avella et al. [2]. They compare thoroughly and favorably their results with the previously best known results in literature. As most works in the literature, we solve the instances of the Beasly set [3], which are divided into 5 categories, A B C D and E. We ignore the instances in categories A and B as Avella et al. have done, because they are too easy. In our tables the instances are named ymn, where y designates the category, m the number of agents and n the number of jobs.

The data which interests us is the execution duration of our program, for the full search as well as for the optimality proof. A full search corresponds to the execution of our program without prior knowledge of any feasible solution value. An optimality proof corresponds to showing there are no solutions with a lower value than the best known. In other words, given a best known solution value z^* (as reported in the literature), to prove optimality our program needs to solve the decision problem $GAP(z^*-1)$, and Avella et al. need to explore their branch-and-cut search tree using z^*-1 as a cutoff value.

The computer used by Avella et al. is a Pentium IV CPU clocked at 3.2 GHz. For our experiments, we had at our disposal a group of cluster nodes all equipped with identical dual AMD Opteron 246 processors. Since Avella et al. also provide the computation time to solve the problems with the CPLEX solver (version 10.1), we did the same to conservatively estimate our machines to be twice as fast as theirs, which concurs with number crunching benchmarks published on a serious computer hardware website (specifically: tomshardware.com). Our algorithm is coded in the C language, but since we use C++ code from [B]TT, we compiled it all with g++ version 4.1 with all optimizations enabled and targeting a x86-64 architecture. Our program executions were all limited to 24 hours user time.

5.3 Results

Tables 1 2 and 3 present the best lower bounds computed by our program for each instance. The initial lower bound is obtained when solving the lagrangian dual in order to determine the initial value of $\bar{z}$. The two next columns list the global lower bounds on the optimal value obtained after 30 minutes and 24 hours of execution time of our method. If no value is indicated in these columns, then the program found an optimal solution before the corresponding time limit. These optimal values are listed in 'Optimal', and are highlighted in bold if they were previously unknown. Otherwise, for the instances which were not solved to optimality in 24 hours, we provide the best known feasible solution values found in the literature in the next column 'Best known'.

Table 4 presents the execution times in seconds for our program and Avella et al. [2]. The results of Avella et al. are indicated as in [2], i.e. without accounting for the difference in computing power. We removed from the table the 10 instances for which neither our program or theirs could even perform an optimality proof. Nine out of these ten are instances of the D category, characterized by having tight knapsack constraints in which the weights are strongly correlated to the prices. The improvement factor is the ratio of their time over our time, divided by 2 to conservatively allow for the difference in hardware, for both full searches and optimality proofs.

As can be seen, in almost all cases our results are much better than theirs, which were state-of-the-art. We find three previously unknown optimal solutions, and are able to find the optimal solution of an instance in all cases they are able to, save one. Our full searches all perform faster and are better than theirs. There are only five instances for which Avella et al. obtain optimality proofs faster than we do, and there is only one instance for which we fail to find an optimal solution while they succeed.

Table 5 lists the total number of nodes evaluated as well as the total number of lagrangian relaxations solved until finding an optimal solution, for the instances for which this happened within 24 hours.

5.4 Concluding analysis

Interestingly, three of the five instances for which Avella et al. perform better to prove optimality are the largest instances of the C category: c40400, c60900 and c401600. The other instances are e201600, a large instance, and c10200, a small but easy instance. The instances in categories C and E are also less constrained than those in category D having tight knapsack constraints whose coefficients are highly correlated to the assignment prices. As a consequence even finding good feasible solutions for instances in category D can be a challenge, notice however that our method performs relatively well in D.

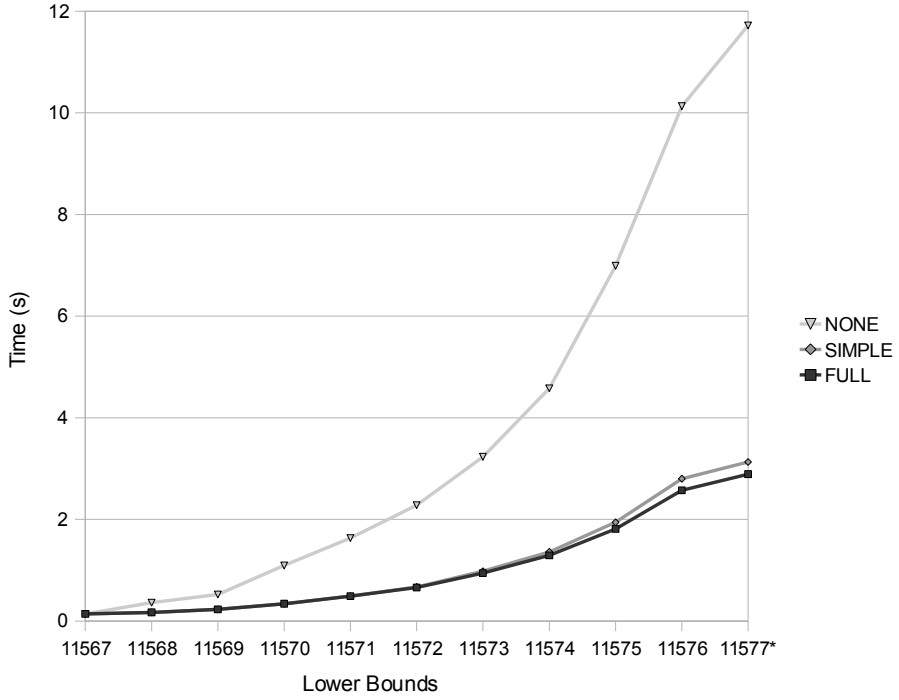

Figure 3: Impact of variable fixing on the time required for instance e10100.

The curves in figure 3 illustrate the benefit of using our variable-fixing scheme. The points on FULL indicate the cumulative time spent for solving the decision problems up to the current value for $\bar{z}$, when using all variable fixation rules presented in section 3. The points on SIMPLE correspond to applying only the so-called simple variable fixation rules, and NONE to applying no variable fixation rules at all.

This illustrates that computing the lagrangian relative costs and applying at least the simple rules is strongly beneficial to the overall performance of the search. Applying the full set of rules incurs little overhead once the relative costs have been computed, and while the improvement over the simple rules is not in the same order of magnitude, it still appears to be worthwile.

The results for this 10-agent 100-job instance are representative of the other instances, for which a similar pattern emerges. The cumulative time also strongly correlates with the cumulative branch-and-bound tree node count as well as the cumulative number of lagrangian relaxations evaluated. This is the case for this instance as well as for the others.

Now let us come back to our clame made in section 2 that our method performs more consistently than more straightforward branch-and-bound methods. We initially implemented a method very similar to that proposed by Haddadi

and Ouzia [6], namely a lagrangian branch-and-bound where an additional repair process is applied at each node to reach feasible solutions and improve the upper bound, to which we added our variable-fixing rules. However, we observed that the performance of such a method is very sensitive to how soon a good feasible solution is found. In particular, we obtained very uneven results for the instances in the D category, where the tight knapsack constraints make it difficult to find feasible solutions.

Figure 4 illustrates this for instance d05100 (5 agents, 100 jobs), the optimum being 6353. We plot the evolution of the upper and lower bounds of three methods: our implementation of Haddadi and Ouzia [6] in light gray, this method with the addition of our variable-fixing rules in dark gray, and the method presented in this paper in black. Notice that the use of variable-fixing rules has hardly any impact on the lower bound in the straighforward branch-and-bound implementation, in contrast to our method presented in this paper (also illustrated previously in figure 3).

Let us conclude with a few additional comments. Solving the instance d05100 with our method without using any variable-fixing rules requires about 18 seconds, and this is consistent with the behavior illustrated in figure 3. Note also that the time reported in table 4 for solving this instance is 5.07 seconds, which is slightly less than depicted in figure 4. The reason for this apparent discrepancy is that we used best-first node selection in figure 4 instead of depth-first.

It may be interesting to find other 0-1 problems for which a scheme similar to ours could be succesfully applied.

Acknowledgements

We wish to thank the associate editor and the anonymous referee of the journal to which we submitted this paper, their comments and suggestions helped to improve its clarity.

References

[1] A. Atamtürk and M.W.P. Savelsbergh. Integer-programming software systems. *Annals of Operations Research*, 140(1):67–124, 2005.

[2] P. Avella, M. Boccia, and I. Vasilyev. A computational study of exact knapsack separation for the generalized assignment problem. *Computational Optimization and Applications*, 45(3):543–555, 2010.

[3] J.E. Beasly. Generalised assignment problem test data sets. `http://people.brunel.ac.uk/~mastjjb/jeb/orlib/gapinfo.html`.

[4] J.A. Diaz and E. Fernandez. A tabu search heuristic for the generalized assignment problem. *European Journal of Operational Research*, 132(1):22–38, 2001.

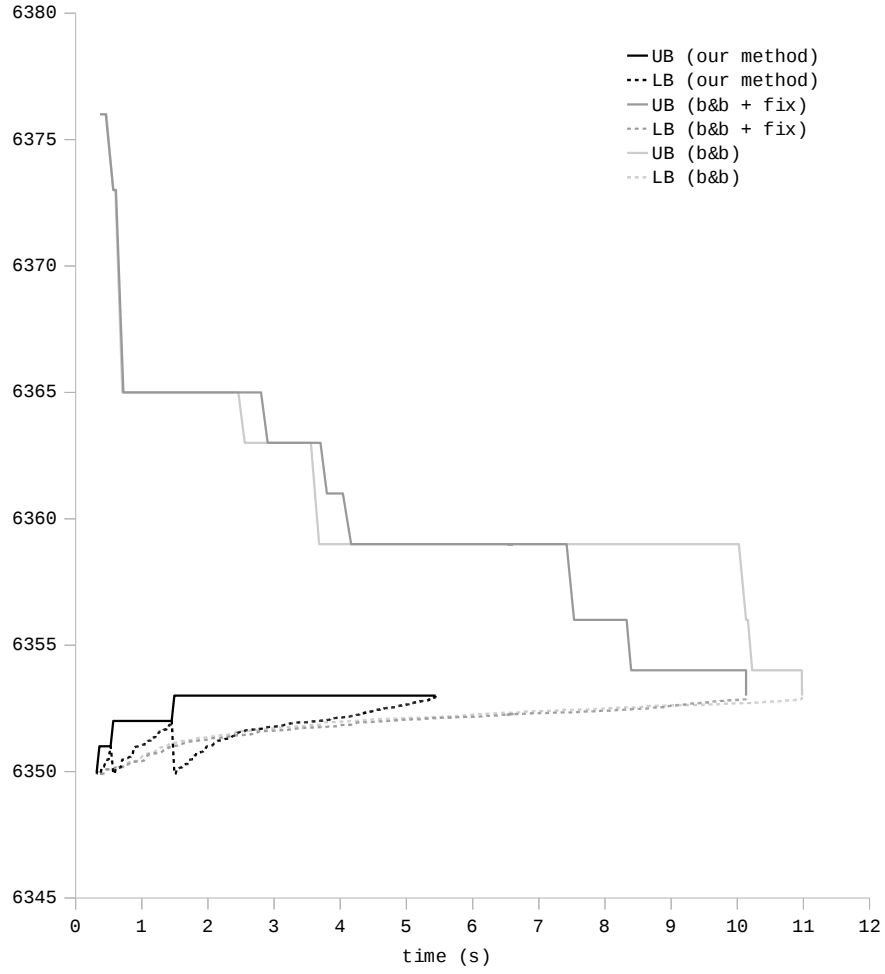

Figure 4: Evolution of bound values for instance d05100.

[5] A. Frangioni. Solving semidefinite quadratic problems within nonsmooth optimization algorithms. *Computers & Operations Research*, 23(11):1099–1118, 1996.

[6] S. Haddadi and H. Ouzia. Effective algorithm and heuristic for the generalized assignment problem. *European Journal of Operational Research*, 153(1):184–190, 2004.

[7] N. Karabakal, J.C. Bean, and J.R. Lohmann. A steepest descent multiplier adjustment method for the generalized assignment problem. Technical re-

port, University of Michigan, 1992.

[8] R.M. Nauss. Solving the generalized assignment problem: An optimizing and heuristic approach. *INFORMS Journal on Computing*, 15(3):249–266, 2003.

[9] A. Pigatti, M.P. de Aragao, and E. Uchoa. Stabilized branch-and-cut-and-price for the generalized assignment problem. *Electronic Notes in Discrete Mathematics*, 5:389–395, 2005.

[10] D. Pisinger. A minimal algorithm for the 0-1 knapsack problem. *Operations Research*, 45(5):758–767, 1997.

[11] G.T. Ross and R.M. Soland. A branch and bound algorithm for the generalized assignment problem. *Mathematical programming*, 8(1):91–103, 1975.

[12] M. Savelsbergh. A branch-and-price algorithm for the generalized assignment problem. *Operations Research*, 45(6):831–841, 1997.

[13] L.A. Wolsey. *Integer programming*. Wiley New York, 1998.

[14] M. Yagiura, T. Ibaraki, and F. Glover. An ejection chain approach for the generalized assignment problem. *INFORMS Journal on Computing*, 16(2):133–151, 2004.

[15] M. Yagiura, T. Ibaraki, and F. Glover. A path relinking approach with ejection chains for the generalized assignment problem. *European journal of operational research*, 169(2):548–569, 2006.

Instance	Initial LB	LB (1/2h+)	LB (24h+)	Optimal	Best known
c05100	1930			1931	
c05200	3455			3456	
c10100	1400			1402	
c10200	2804			2806	
c10400	5596			5597	
c15900	11339			11340	
c20100	1242			1243	
c20200	2391			2391	
c20400	4781			4782	
c201600	18802	18802		**18802**	
c30900	9982			9982	
c40400	4244			4244	
c401600	17144	17144	17145		17145
c60900	9325	9326	9326		9326
c801600	16284	16284	16284		16289

Table 1: Solution values found by our method for C instances.

Instance	Initial LB	LB (1/2h+)	LB (24h+)	Optimal	Best known
d05100	6350			6353	
d05200	12741			12742	
d10100	6342			6347	
d10200	12426			12430	
d10400	24959			**24961**	
d15900	55403	55403	55404		55414
d20100	6177	6184		6185	
d20200	12230	12234	12235		12244
d20400	24561	24562	24563		24585
d201600	97823	97824	97824		97837
d30900	54833	54833	54834		54868
d40400	24350	24350	24350		24417
d401600	97106	97106	97106		97113
d60900	54551	54551	54551		54606
d801600	97034	97034	97034		97052

Table 2: Solution values found by our method for D instances.

Instance	Initial LB	LB (1/2h+)	LB (24h+)	Optimal	Best known
e05100	12673			12681	
e05200	24927			24930	
e10100	11568			11577	
e10200	23302			23307	
e10400	45745			45746	
e15900	102420			102421	
e20100	8432			8436	
e20200	22377			22379	
e20400	44876			44877	
e201600	180644			180645	
e30900	100427			100427	
e40400	44557			44561	
e401600	178292			178293	
e60900	100147	100148		100149	
e801600	176819			**176820**	

Table 3: Solution values found by our method for E instances.

	Our program		Avella et al.		Improvement factor	
Instance	Search	Proof	Search	Proof	Search	Proof
c05100	.05	.04	2.39	.60	23.9	7.5
c05200	.42	.36	9.03	1.50	10.8	2.1
c10100	.16	.13	2.67	.80	8.3	3.1
c10200	3.12	2.28	17.13	2.90	2.7	.6
c10400	4.54	2.65	38.61	5.70	4.3	1.1
c15900	17.27	2.70		2,257.37		418.0
c20100	.29	.17	1.83	.80	3.2	2.4
c20200	.81	.17	13.98	1.80	8.6	5.3
c20400	25.32	9.79	91.52	21.90	1.8	1.1
c201600	5,804.55	3.52				
c30900	373.27	3.35	3,997.56	12.30	5.4	1.8
c40400	9.94	2.87	52.38	3.30	2.6	.6
c401600		11,831.26		3,231.14		.1
c60900		203.99	8,369.69	121.40		.3
d05100	5.07	1.28	17.08	13.30	1.7	5.2
d05200	2.33	1.24	17.83	13.80	3.8	5.6
d10100	41.33	13.37		385.29		14.4
d10200	1,144.67	417.02		8,921.75		10.7
d10400	513.14	174.43				
d20100	4,010.77	2,425.81		27,783.04		5.7
e05100	1.05	.28	5.33	2.00	2.5	3.6
e05200	.59	.23	5.69	1.50	4.8	3.3
e10100	2.92	.90	8.94	4.70	1.5	2.6
e10200	4.91	2.92	39.41	7.40	4.0	1.3
e10400	.86	.56	98.91	3.50	57.5	3.1
e15900	3.58	.95	203.45	6.60	28.4	3.5
e20100	2.53	1.21	51.14	8.30	10.1	3.4
e20200	.98	.65	42.89	5.80	21.9	4.5
e20400	1.97	.94	79.89	8.10	20.3	4.3
e201600	40.02	35.80	1,003.47	32.70	12.5	.5
e30900	4.56	.77	671.51	13.20	73.6	8.6
e40400	104.59	56.58	2,875.01	636.60	13.7	5.6
e401600	243.02	3.69	4,123.61	54.60	8.5	7.4
e60900	23,181.30	1,360.24		6,341.15		2.3
e801600	75.51	12.05				

Table 4: Performance.

Instance	Nodes	Relaxations
c05100	6	397
c05200	11	2,112
c10100	31	1,543
c10200	248	17,970
c10400	157	13,898
c15900	406	26,894
c20100	50	2,226
c20200	52	3,689
c20400	1,034	63,210
c201600	62,630	3,759,201
c30900	6,826	369,235
c40400	253	16,626
d05100	617	36,848
d05200	159	10,535
d10100	3,824	191,735
d10200	58,840	3,346,652
d10400	19,577	1,157,941
d20100	379,632	13,857,852
e05100	231	14,013
e05200	82	5,389
e10100	425	23,040
e10200	441	27,903
e10400	27	2,385
e15900	67	4,573
e20100	207	10,788
e20200	34	2,662
e20400	79	4,071
e201600	386	24,803
e30900	71	3,743
e40400	3,223	143,889
e401600	1,631	84,549
e60900	210,933	9,519,630
e801600	285	14,230

Table 5: Node and lagrangian relaxation evaluation count.