ORiON, Vol. 19, No. 1/2, pp 33-52 *ISSN 0259-191-X*

A PARTITIONING SCHEME FOR SOLVING THE 0-1 KNAPSACK PROBLEM

M.F. KRUGER and J.M. HATTINGH
School of Computer, Statistical and Mathematical Sciences
North-West University (Potchefstroom Campus)
Private Bag X6001, Potchefstroom, 2520, South Africa
E-mail: bwimfk@puknet.puk.ac.za, rkwjmh@puknet.puk.ac.za

ABSTRACT
The application of valid inequalities to provide relaxations which can produce tight bounds, is now common practice in Combinatorial Optimisation. This paper attempts to complement current theoretical investigations in this regard. We experimentally search for "valid" equalities which have the potential of strengthening the problem's formulation.

Recently, Martello and Toth [13] included cardinality constraints to derive tight upper bounds for the 0-1 Knapsack Problem. Encouraged by their results, we partition the search space by using equality cardinality constraints. Instead of solving the original problem, an equivalent problem, which consists of one or more 0-1 Knapsack Problem with an exact cardinality bound, is solved.

By explicitly including a bound on the cardinality, one is able to reduce the size of each subproblem and compute tight upper bounds. Good feasible solutions found along the way are employed to reduce the computational effort by reducing the number of trees searched and the size of the subsequent search trees.

We give a brief description of two Lagrangian-based Branch-and-Bound algorithms proposed in Kruger [9] for solving the exact cardinality bounded subproblems and report on results of numerical experiments with a sequential implementation. Implications for and strategies towards parallel implementation are also given.

Keywords: Search Algorithms, Combinatorial Optimisation, Mathematical Programming.

1. INTRODUCTION AND MOTIVATION

Let *n items* with *weights* a_1, a_2, ..., a_n, and c_1, c_2, ..., c_n be given. The *Knapsack Problem* is to pack a given knapsack with *capacity b* so as to maximize the total profit. The 0-1 *Knapsack Problem* may thus be formulated as

$$z_{IP} = \max\left\{\sum_{j=1}^{n} c_j x_j \,\middle|\, \sum_{j=1}^{n} a_j x_j \le b, \quad x_j \in \{0,1\} \ \text{ for } j = 1,2,\ldots,n\right\}, \tag{1}$$

where all data are integers, and $x_j = 1$ if item j is included in the knapsack and 0 otherwise.

Without loss in generality we may assume that the $a_j > 0$, $c_j > 0$, $a_j \le b$, for $j = 1, 2, ..., n$ and $\sum_{j=1}^{n} a_j > b$.

An upper bound $\bar{z}$, for z_{IP}, can be produced by relaxing the integrality constraints on the variables x_j and solving the Linear Programming relaxation problem and setting $\bar{z} = \lfloor z_{LP} \rfloor$, where $\lfloor z \rfloor$ is the largest integer $\le z$. If the weights and profits are sorted according to non-increasing *profit density*, i.e.

$$\frac{c_j}{a_j} \ge \frac{c_{j+1}}{a_{j+1}}, \quad \text{for } j = 1,2,\ldots,n-1, \tag{2}$$

and

$$\sum_{j=1}^{t-1} a_j \le b < \sum_{j=1}^{t} a_j, \tag{3}$$

then the LP solution is given by Dantzig [4]

$$x_j = 1, \ \text{for } j = 1,2,\ldots,t-1$$
$$x_j = 0, \ \text{for } j = t+1,\ldots,n, \ \text{and}$$
$$x_t = \left(b - \sum_{j=1}^{t-1} a_j\right) \Big/ a_t,$$

while

$$z_{LP} \equiv \sum_{j=1}^{t-1} c_j + \frac{c_t}{a_t}\left(b - \sum_{j=1}^{t-1} a_j\right). \tag{4}$$

We will call the above LP-solution with x_t set to 0, the *break solution* and t the *break item*. The bound $\bar{z}$ is called the *Dantzig-bound*. If the items are already sorted the computation of z_{LP} is done in $O(n)$ time.

1.1 Historical Notes on Exact Algorithms

The 0-1 Knapsack Problem has received considerable attention, not only because it has several important applications in itself, but also as a substructure in many discrete

optimisation problems. The techniques for *exactly* solving the 0-1 Knapsack Problem can be classified into three groups: Dynamic Programming based algorithms, Branch-and-Bound based algorithms, and hybrid approaches.

Early papers which specialized *Dynamic Programming* for the 0-1 Knapsack Problem include Bellman [2], Dantzig [4] and Bellman and Dreyfus [3]. The idea is to first fill a small knapsack optimally and then, using this information, fill larger a knapsack optimally. This process is repeated until the original problem is solved completely. Some computational improvements were proposed by Toth in [17].

In his award winning PhD-thesis Pisinger [16, Chapter 4] devised a dynamic programming recursion, which, although the worst-case time complexity is still $O(bn)$ as for the Bellman recursion, solves most relatively large problem instances without enumerating too many variables. The algorithm starts from the break solution and at each stage either inserts or removes an item. Strong upper bounds are used to limit the number states in the recursion. The enumeration process terminates due to some bounding tests, in which case it is possible to prove that the current incumbent solution is optimal.

Recently, Martello, Pisinger and Toth [10] incorporated cardinality constraints into a very efficient Dynamic Programming algorithm. Although the worst-case time complexity of their algorithm is still $O(bn)$, they solved most instances quite quickly due to the tight bounds produced by the cardinality constraints.

One of the first *Branch-and-Bound* algorithms for the 0-1 Knapsack Problem was proposed by Kolesar [8]. It was a straightforward specialization of the Land and Doig algorithm for general integer programming problems. The algorithm used the best bound selection rule and branching was done on the fractional variable. The large computer memory requirements of this algorithm led to the development of other Branch-and-Bound algorithms by Horowitz and Sahni [7], Nauss [14], Fayard and Plateau [6] and Martello and Toth [11], to name but a few. These algorithms use a depth-first selection rule and branching is done on the free variable with the smallest index (assuming that the items are sorted according to non-increasing profit densities). The depth-first enumeration limits the space consumption of these algorithms. Many of these algorithms include a reduction or preprocessing phase in which some of the extremal variables are fixed to 0 or 1.

The performance of a Branch-and-Bound algorithm very much depends on the tightness and application of upper bounds. Several bounds were proposed in the literature of which Martello and Toth [11, 12, 13] were some of the main contributors. Bounds are generally based on Lagrangian relaxation, partial enumeration, the adding of valid inequalities, and relaxations of these valid inequalities.

Martello and Toth [11] compared some of these algorithms. Their results showed that "easy" problem instances can be solved efficiently.

Hybrid approaches to the solving of the 0-1 Knapsack Problem are mainly devised by combining Branch-and-Bound with Dynamic Programming. One approach is to do a Branch-and-Bound search up to a certain time limit and thereafter automatically switch over to Dynamic Programming (see Dudziński and Walukiewicz [5]). An approach recently proposed by Martello and Toth [13] , is to do a Branch-and-Bound search down to a certain level in the search tree or while the residual capacity is above a certain threshold value. Dynamic Programming is then used to solve the subproblem to optimality.

1.2 Aspects of Search Models

In order to avoid enumerating all feasible solutions a good *search strategy* and a *bounding procedure* are crucial. Bounding procedures are usually based on the exact solution of a *relaxed problem* which is obtained from the original problem by relaxing or dropping some of the constraints. The application of *valid inequalities* to provide relaxations which can produce tight bounds, is now common practice. With this approach one tries to find a polyhedral description of the convex hull of the feasible points. This paper attempts to complement current theoretical investigations in this regard. We experimentally search for "valid" equalities which have the potential of strengthening the problem's formulation. The inclusion of these equalities in the problem formulation partitions the search space.

If one partitions the search space before it is searched and concentrate the search only on those parts that have the potential of delivering feasible solutions better than the current best, one has more than one search tree. The best solution found amongst these trees (forest search) is then the optimal solution of the original problem. This approach lends itself naturally to a high-level parallel implementation in which each processing node is given a tree to search. It

is a well-known fact that the branching in a search tree can be reduced significantly if tight lower and upper bounds are known. If we partition the search space into *N* different parts, the formulation of the original problem must be specialized for each different part by adding extra constraints. In general, this will lead to tighter upper bounds. The maximum of the upper bounds taken over all parts of the partition will produce a global upper bound which, in general, will be tighter than upper bounds derived for the original problem.

The availability of a tight global upper bound early in the solution process has the added advantage that the "quality" of any incumbent solution (the best feasible found so far) can be appraised much more reliably. This means that if one wants to curtail the search one has a better estimate of how "good" the incumbent solution really is. This is the main reason why we chose the Branch-and-Bound algorithm and not Dynamic Programming. One of the drawbacks of Dynamic Programming is that a solution is only available at the end of the process.

Recently, Martello and Toth [13] included cardinality constraints (i.e. on the number of items in the knapsack) to derive tight upper bounds in their Branch-and-Bound algorithm. Encouraged by their results, we partition the search space by using *equality* cardinality constraints. So, instead of solving the original problem an equivalent problem which, in general, consists of one or more separate problems, is solved. Each separate problem is a 0-1 Knapsack Problem with an exact bound (an equality constraint) on the cardinality. These subproblems will be called *Exact k-item* 0-1 *Knapsack Problems*. Pandit and Ravi Kumar [15] used a similar approach for the solution of strongly-correlated knapsack problem instances.

By explicitly including a bound on the cardinality, one is able to reduce the size of each subproblem and compute tight upper bounds. Furthermore, if a good feasible solution is found along the way it may reduce the computational effort by reducing the number of trees searched and the size of the subsequent search trees.

In the following section we will illustrate some of the properties of the cardinality constrained 0-1 Knapsack Problem. In Section 3 we define an equivalent problem and give a high-level description of our partitioning scheme. The inclusion of the extra equality constraint can be handled by using Lagrangian Relaxation or by using an LP-solver to provide upper bounds in a Branch-and-Bound type algorithm. In order to compete favourably with current exact

algorithms for the 0-1 Knapsack Problem, one needs an algorithm that is efficient. Section 4 gives a brief description of two Lagrangian-based algorithms, proposed in Kruger [9], for solving the Exact *k*-item 0-1 Knapsack Problems. Some techniques for reducing the size of the subproblems are also given. Computational experiments are presented in Section 5, followed by a conclusion.

2. CARDINALITY CONSTRAINED KNAPSACK PROBLEMS

2.1 The Maximum *k*-item Knapsack Problem

Assume that the items are sorted according to non-decreasing *weights*, i.e. $a_j \le a_{j+1}$, for j = 1, 2, ..., n - 1. Suppose that we fill our knapsack according to non-decreasing weights, i.e.

$$\sum_{j=1}^{k_u} a_j \le b < \sum_{j=1}^{k_u+1} a_j, \tag{5}$$

then it is clear that in any integer solution to (1), the number of items in our knapsack is bounded from above by k_u and hence we may introduce the canonical inequality

$$\sum_{j=1}^{n} x_j \le k_u. \tag{6}$$

Note that k_u can be found without sorting by using a partitioning scheme similar to the one used by Balas and Zemel [1] to find the break item.

Definition 1 (Maximum *k*-item 0-1 Knapsack Problem) *Given $k = k_u$, we define the* Maximum *k*-item 0-1 Knapsack Problem (*MCP*) *as follows*

$$\begin{aligned} & z_{MCP}(k_u) = \max \sum_{j=1}^{n} c_j x_j \\ \text{subject to } & \sum_{j=1}^{n} a_j x_j \le b, \\ & \sum_{j=1}^{n} x_j \le k_u, \\ & x_j \in \{0,1\} \quad \text{for } j = 1,2,\ldots,n. \end{aligned} \tag{7}$$

Solving the continuous case of the above problem using an LP-solver can be computationally expensive. Martello and Toth [13] solved the continuous relaxation of (7) without an LP-solver by using Lagragian relaxation to add the cardinality constraint to the objective function.

More recently, Martello *et al*. [10] solved the continuous case of (7) by surrogate relaxing the cardinality constraint with the knapsack constraint. Since we have only two constraints, we can limit the explicit number of surrogate multipliers to one multiplier. Martello *et al.* [10] prove some monotonicity properties and derive a special binary search, similar to that of Martello and Toth [13], which considers only a limited number of integer surrogate multipliers. They report that since the continuous bounds are generally tight, the transformed problem tends to be solved much easier. According to them, their approach has the additional advantage that if the optimal solution to the LP-relaxed problem is correct, i.e. the cardinality constraint is satisfied, one also obtains a feasible solution to the original problem, thus solving the problem to optimality.

2.2 The Minimum *k*-item Knapsack Problem

Assume that the items are sorted according to non-increasing *profits*, i.e. $c_j \geq c_{j+1}$, for $j = 1, 2, \ldots, n-1$. Suppose that z_{best} is our current best lower bound for the 0-1 Knapsack Problem and that we fill our knapsack according to non-increasing profits, i.e.

$$\sum_{j=1}^{k_l - 1} c_j \leq z_{best} < \sum_{j=1}^{k_l} c_j, \tag{8}$$

then it is clear that in any integer solution to the 0-1 Knapsack Problem with solution value better than z_{best}, the number of items in our knapsack is bounded from below by k_l and hence we may introduce the canonical inequality

$$\sum_{j=1}^{n} x_j \geq k_l. \tag{9}$$

Note again that k_l can be found without sorting by using a partitioning scheme similar to the one used by Balas and Zemel [1] to find the break item.

Definition 2 (Minimum *k*-item 0-1 Knapsack Problem) *Given $k = k_l$, we define the* Minimum *k*-item 0-1 Knapsack Problem *as follows*

$$\begin{aligned} z_{NCP}(k_l) &= \max \sum_{j=1}^{n} c_j x_j \\ \text{subject to } & \sum_{j=1}^{n} a_j x_j \leq b, \\ & \sum_{j=1}^{n} x_j \geq k_l, \\ & x_j \in \{0,1\} \quad \text{for } j = 1, 2, \ldots, n. \end{aligned} \tag{10}$$

Martello and Toth [13] and Martello *et al.* [10] solved the continuously relaxed version of problem (10) with algorithms similar to the ones used to solve the Maximum *k*-item 0-1 Knapsack Problem and the reader is referred to the relevant articles and technical reports for details.

In our partitioning scheme we solve the closely related cardinality constrained 0-1 Knapsack Problem, in which the constraint is an *equality* constraint, repeatedly.

2.3 The Exact *k*-item Knapsack Problem

Suppose that we know beforehand that the number of items in an optimal solution to the 0-1 Knapsack Problem is, k_e, then we can solve the original problem by solving the following equivalent problem.

Definition 3 (Exact *k*-item 0-1 Knapsack Problem) *Given* $k = k_e$*, we define the* Exact *k*-item 0-1 Knapsack Problem (*EKP*(*k*)) *as follows*

$$
\begin{aligned}
& z_{EKP}(k_e) = \max \sum_{j=1}^{n} c_j x_j \\
\text{subject to } & \sum_{j=1}^{n} a_j x_j \le b, \\
& \sum_{j=1}^{n} x_j = k_e, \\
& x_j \in \{0,1\} \quad \text{for } j = 1,2,\ldots,n.
\end{aligned}
\tag{11}
$$

if the solution exists, otherwise $z_{EKP}(k_e) = 0$. We denote the LP relaxation of EKP(k) by LEKP(k) and the corresponding objective value by $z_{LEKP}(k)$.

Kruger [9] recently proposed two Lagragian-based Branch-and-Bound algorithms which solve problem (11) exactly (see Section 4 for a brief description).

3. A PARTITIONING SCHEME

It is now easy to see that one can solve the original 0-1 Knapsack Problem by solving *EKP*(*k*) for each *k* in the cardinality range, $\Re$. This gives rise to the following equivalent formulation:

Definition 4 (The Equivalent Problem) *Let* $x_{best} = \left(x_j^{best}\right)$ *be any feasible solution for the* 0-1 *Knapsack Problem and* $\Re \neq \varnothing$ *the associated cardinality range. Then solving the* 0-1 *Knapsack Problem is equivalent to solving the following problem*:

$$z_{IP} = \max\left\{\max_{k\in\Re} z_{EKP}(k),\, z_{best}\right\}, \tag{12}$$

where $z_{best} = \sum_{j=1}^{n} c_j x_j^{best}$.

We are now ready to state our algorithm.

3.1 An Exact Algorithm for the 0-1 Knapsack Problem

Given an algorithm, **`ECardKnap`**, which can solve any instance of the Exact *k*-item 0-1 Knapsack Problem exactly, the algorithm is as follows:

Algorithm 1 `CardKnap`

1: Determine the LP-solution for the original (unsorted) problem by using the partitioning scheme of Balas and Zemel [1];
2: Construct an incumbent solution with solution value, z_{best};
3: Try to fix some variables at their upper or lower bounds, by using reduced costs;
4: Sort the reduced problem according to non-increasing profit densities;
5: Compute the break solution, x^b, for the reduced problem;
6: Try and improve on z_{best} by filling the remaining capacity in a greedy fashion;
7: Reduce the problem by using probing to fix some variables at 0 or 1;
8: Call **`CardRange`** to calculate the cardinality range, $\Re$, for the reduced problem and $g(k) := \lfloor z_{LEKP}(k) \rfloor$ for all $k \in \Re = \{k_1, \ldots, k_2\}$;
9: **for** $k = k_1$ to k_2 **do**
10: **if** $\left(g(k) \le z_{best}\right)$ **then**
11: **cycle**;
12: **end if**
13: Solve the LP-relaxation of EKP(k) to get $z_{LEKP}(k)$;
14: Try to reduce the subproblem by using reduced costs (and z_{best} as lower bound) to fix some variables at 0 or 1;
15: Compute $z_{EKP}(k)$ by calling **`ECardKnap`** to solve the reduced problem;
16: **if** $z_{EKP}^k > z_{best}$ **then**
17: $z_{best} := z_{EKP}^k$ (and save new solution);
18: **if** z_{best} = *current upperbound*, STOP;
19: **end if**
20: **end for**

3.2 Computing the *Cardinality Range*

We need the following definition and lemma to devise an algorithm to compute the cardinality range.

Definition 5 *Define* $z_{LEKP}(k): \mathbb{Z}^+ \to \mathbb{Z}^+$ *by*

$$\begin{aligned} z_{LEKP}(k) &= \max \sum_{j=1}^{n} c_j x_j \\ \text{subject to } & \sum_{j=1}^{n} a_j x_j \le b, \\ & \sum_{j=1}^{n} x_j = k, \\ & 0 \le x_j \le 1 \quad \text{for } j = 1, 2, \ldots, n. \end{aligned} \tag{13}$$

if the solution exists, otherwise $z_{LEKP}(k) = 0$.

Definition 6 (Quasi-concave Function) *A function* $g: \Im \to \mathbb{Z}$ *is called* quasi-concave *over* $\Im$ *if, for any two points* $k_1, k_2 \in \Im$ *such that* $g(k_1) \le g(k_2)$ *and for all* $\lambda \in [0, 1]$,

$$g(k_1) \le g\left(\lambda k_1 + (1-\lambda) k_2\right), \tag{14}$$

where $\Im$ *is convex*.

The proof of the following Lemma is given in Kruger [9].

Lemma 7 *Suppose that* $z_{LEKP}(k_1)$ *and* $z_{LEKP}(k_2)$ *exists, where* $k_1, k_2 \in \mathbb{Z}^+$ *and* $k_1 < k_2$. *Then* $z_{LEKP}(k)$ *exists for all* $k \in [k_1, k_2]$ *and* $g: [k_1, k_2] \to \mathbb{Z}$ *defined by*

$$g(k) := \lfloor z_{LEKP}(k) \rfloor, \tag{15}$$

is a quasi-concave *function over* $[k_1, k_2]$.

Note that Lemma 7 is in general not true for the case where $g(k)$ is replaced by $g(k) := z_{EKP}(k): \mathbb{Z} \to \mathbb{Z}$.

Definition 8 (Cardinality Range) *Let* z_{best} *be a lower bound for* z_{IP}. *We define the* cardinality range, $\Re$, *for the* 0-1 *Knapsack Problem by*

$$\Re = \{k_1, k_1 + 1, \ldots, k_2\} = \left\{k \in \mathbb{Z} \;\middle|\; \lfloor z_{LEKP}(k) \rfloor > z_{best}\right\}. \tag{16}$$

If x^b is the current incumbent solution, let $z_{best} = \sum_{j=1}^{n} c_j x_j^b$ and $k' = \sum_{j=1}^{n} x_j^b$. The *cardinality range*, $\Re$, can be computed by **Algorithm 2** which first tries to find the largest $k_2 \in \{1, 2, ..., k'\text{-}1\}$ searching from right to left, such that $g(k_2) := \lfloor z_{LEKP}(k_2) \rfloor > z_{best}$. If it finds such a k_2, it continues until it find the largest $k_1 \in \{1, 2, ..., k_2\text{-}1\}$ such that $g(k_1 - 1) := \lfloor z_{LEKP}(k_1 - 1) \rfloor \le z_{best}$. Otherwise, it tries to find the smallest $k_1 \in \{k'+1, \ldots, n\}$ searching from left to right, such that $g(k_1) := \lfloor z_{LEKP}(k_1) \rfloor > z_{best}$. If it finds such a k_1, it continues until it find the smallest $k_2 \in \{k_1+1, ..., n\}$ such that $g(k_2 + 1) := \lfloor z_{LEKP}(k_2 + 1) \rfloor \le z_{best}$.

Algorithm 2 `CardRange`

Require: z_{best} and k'
Ensure: Cardinality range, $\Re = \{k_1, ..., k_2\}$
Initialize $g(k') := z_{best}$, $k_1 := 0$ and $k_2 := 0$;
for $k := k'\text{-}1$ down to 1 **do**
  Calculate $g(k) := \lfloor z_{LEKP}(k) \rfloor$;
  if $(g(k) > g(k+1)$ and $g(k+1) = z_{best})$ **then**
    $k_2 := k$;
  else if $(g(k) < g(k+1)$ and $g(k) \le z_{best})$ **then**
    if $(g(k+1) > z_{best})$ **then** $k_1 := k + 1$;
    if $(k_2 > 0)$ **then return**;
      exit for-loop;
  end if
end for
for $k := k' + 1$ to n **do**
  Calculate $g(k) := \lfloor z_{LEKP}(k) \rfloor$;
  if $(g(k-1) < g(k)$ and $g(k-1) = z_{best})$ **then**
    $k_1 := k$;
  else if $(g(k-1) > g(k)$ and $g(k) \le z_{best})$ **then**
    $k_2 := k$ - 1;
    return;
  end if
end for

The time complexity of the above algorithm is dominated by the calculation of $z_{LEKP}(k)$, but recall that we use a tailor-made LP solver to solve it.

3.3 Upper Bounds

If

$$\overline{z}_{range} = \max_{k \in \Re} z_{LEKP}(k) \tag{17}$$

then, in general, $\overline{z}_{range}$ is a tight upper bound which has the potential of reducing the integrality gap.

Computational experience has shown that the cardinality range for most of the data instances is relatively small, the only exception being the subset-sum problem (see Section 5 for definition). Furthermore, if the solution of $z_{EKP}(k)$, for a specific value of k, produces a better lower bound than z_{best}, the cardinality range and z_{best} can be updated.

4. SOLVING THE EXACT *k*-ITEM 0-1 KNAPSACK PROBLEM

Algorithm 1 makes use of an algorithm, **`ECardKnap`**, which can efficiently solve any instance of the Exact k-item 0-1 Knapsack Problem exactly. Kruger [9], recently, proposed two such algorithms. The first algorithm Kruger [9, Chapter 6] uses Lagrangian relaxation to add the knapsack constraint to the objective function. A specialized iterative, but finite and exact, technique is used for determining the optimal Lagrange multipliers.

The second algorithm is an extension of the first. The partitioning idea is taken a step further by introducing an equality constraint on the number of items included in the knapsack, that are not part of the break solution. As a result of this, Kruger was able to derive tight upper bounds and reduce some problem instances even further.

4.1 Problem Reduction

For each specific cardinality, k, one can do a reduction step in which the reduced costs of the LP-solution are used to fix some extremal variables to 0 or 1. Using general LP-solvers is not advised for solving the LP relaxed problem, because of the special structure of the linear programming relaxation of EKP. Since there are only two constraints, the inverse of a basis, for example, can be written down explicitly. We have implemented a tailor-made revised simplex LP-solver (phase 2) that uses this (see Kruger [9] for details).

4.2 Constructing a Feasible Solution

The proof of the following proposition is obvious.

Proposition 9 *Let* x^* *be any basic feasible solution to the linear programming relaxation of the Exact* k*-item* 0-1 *Knapsack Problem. Then*

(i) x^* *has either* 0 *or* 2 *fractional components.*

(ii) *If* x_i^* *and* x_j^* *are fractional, then* $x_i^* + x_j^* = 1$.

(iii)*If* x_i^* *and* x_j^* *are fractional such that* $a_i \le a_j$, *then* $a_i \le b_{red} \le a_j$,

where

$$b_{red} = b - \sum_{r \in N \setminus \{i,j\}} a_r x_r^* \tag{18}$$

and $N = \{1,2,...,n\}$.

This proposition can be used in the following way; one can solve LEKP(k) and note that if we set $x_i = 1$ and $x_j = 0$ in (iii) of the above proposition and adjust the slack accordingly, we get an integer feasible solution. Moreover, sometimes it is possible to construct an even better integer feasible solution by setting $x_r = 1$ where

$$c_r = \max_{i \in N \setminus \{j\}} \left\{ c_i : a_i \le b_{red}, x_i^* = 0 \right\}. \tag{19}$$

In our algorithm we use this technique repeatedly to construct our first incumbent solution.

5. EXPERIMENTAL RESULTS

In this section we present some results of an experimental testing of the ideas proposed in this paper. For a more detailed report see Kruger [9].

5.1 Data Generation

The data instances used in the literature to test and compare knapsack algorithms are usually classified as

(i) *Uncorrelated* (uc): weights a_j and profits c_j are uniformly distributed in [1, R].

(ii) *Subset-sum* (ss): weights a_j are uniformly distributed in [1, R], and profits are set to $c_j = a_j$.

(iii) *Weakly correlated* (wc): weights a_j are uniformly distributed in [1, R], and profits c_j in $[a_j - \delta, a_j + \delta]$ such that $c_j \ge 1$.

(iv) *Strongly correlated* (sc): weights a_j are uniformly distributed in [1, R], and profits are set to $c_j = a_j + \delta$.

(v) *Inverse strongly correlated* (isc): profits c_j are uniformly distributed in [1, R], and weights are set to $a_j = c_j + \delta$.

(vi) *Almost strongly correlated* (asc): weights a_j are uniformly distributed in [1, R], and profits c_j in [$a_j + 0.99\delta$, $a_j + 1.01\delta$].

(vii) *Uncorrelated instances with nearly similar weights* (ucsw): weights a_j are uniformly distributed in [100R, 100.1R], and profits c_j in [1, R].

Of these classes, the strongly correlated (sc), almost strongly correlated (asc) and inverse strongly correlated (isc) classes are regarded by many researchers in this field as the *difficult* classes.

In the results that follow we do not report on uncorrelated instances with nearly similar weights (ucsw) instances, because the integers involved for problems of high dimension could not be handled by our (current) computer code.

We used an algorithm (similar to that given in Pisinger [16, pages 105-106]) to generate test instances. The algorithm uses the standard **`lrand48`** generator of the C library for generating pseudo-random profits and weights. A seed for the algorithm is given by the C library procedure **`srand48`**.

5.2 Experiments

The proposed algorithm, **Algorithm 1**, was coded in FORTRAN90 and tested on an IBM RISC6000 with 128Mb of memory. We did one set of tests by using the partition-based algorithm proposed in Kruger [9, Chapter 7] to solve the resulting Exact k-item 0-1 Knapsack Problems.

Each value in the tables and graphs, shown below, is the average taken over 10 problem instances (*seed* = 100, 200, ..., 1000). The values in the tables given between the parentheses is the standard deviation for these 10 problems.

5.3 Results

(a) The reduction by the first reduction step was very good for the "easy" problem instances but poor for the "difficult" instances. For the subset-sum instances this reduction should be zero, but for large problems it often happens that the constructed greedy solution is optimal.

(b) The reduction by the second reduction step was good for almost all problem instances. This is partly due to the fact that good solutions are often found during the reduction process itself.

(c) The reduction by the third reduction step was good for almost all problem instances. For the subset-sum and strongly correlated instances this reduction should be zero, but for large problems it often happens that an optimal greedy solution can be constructed. What often happens in this case is that as soon as the maximum (equality) cardinality constraint is enforced, an optimal solution is produced.

(d) Efforts to reduce the integrality gap by the inclusion of cardinality constraints produced very strong upper bounds for "difficult" problem instances (*sc*, *isc*, *asc*), in Fig. 1. It, therefore, seems that putting more effort into obtaining better feasible solutions is the only way to further improve the performance in these instances.

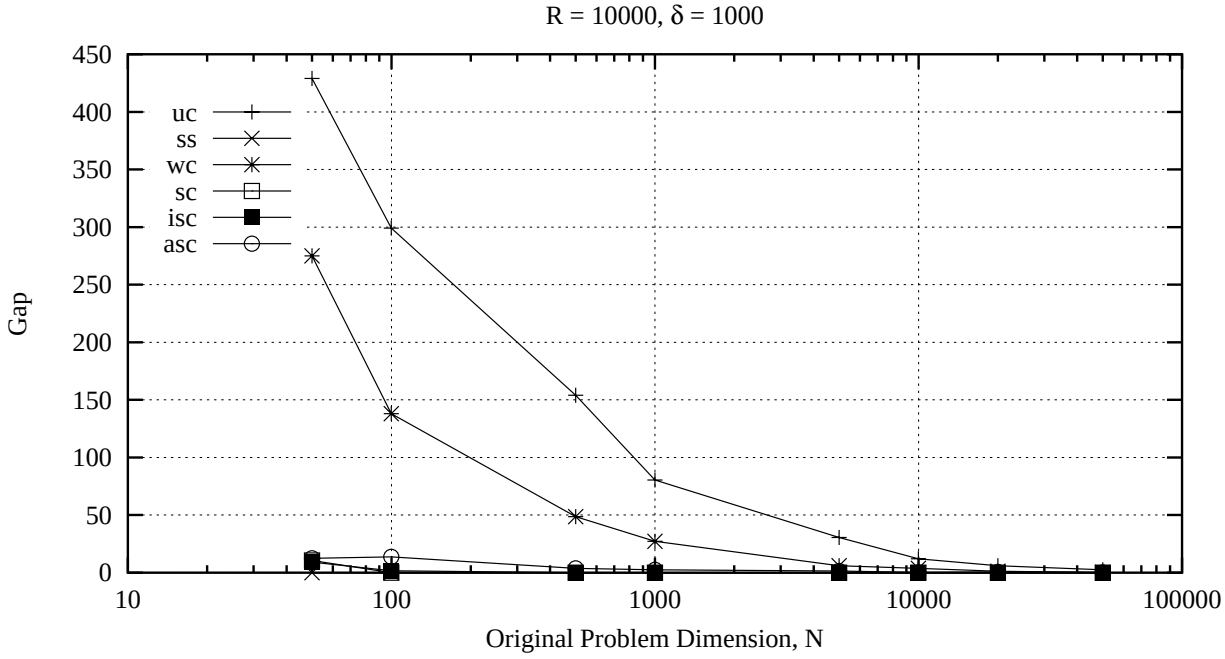

Figure 1: Integrality gap after **`CardRange`** (R = 1000, δ = 1000)

(e) The proposed algorithm lends itself naturally to a high-level parallel implementation. Table 1 give an upper bound on the (average) number of trees that have to be searched (potentially), while Table 2 report on the number of trees actually searched.

Table 1: Number of cardinalities in Cardinality Range (R = 1000, δ = 1000)

Original Size	50	100	500	1000	5000	10000	20000	50000
uc	2.60(1.02)	3.60(1.80)	5.20(0.98)	6.50(1.80)	7.80(1.94)	8.70(2.90)	13.4(2.7)	11.6(5.6)
ss	18.7(6.6)	40.7(2.3)	187.(62.)	329.(165.)	828.(1014.)	1653.(2025.)	828.(2483.)	0.00(0.00)
wc	4.50(0.67)	4.90(1.45)	8.50(1.91)	9.90(2.74)	15.3(4.6)	16.0(7.0)	19.5(6.4)	27.3(6.6)
sc	1.00(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)
isc	1.00(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)
asc	1.10(0.30)	1.00(0.00)	1.00(0.00)	1.10(0.30)	1.00(0.00)	1.00(0.00)	1.00(0.00)	1.20(0.40)

Note: A zero value in the table means that all the problems were solved prior to the cardinality calculations.

Table 2: Number of trees searched (R = 1000, δ = 1000)

Original Size	50	100	500	1000	5000	10000	20000	50000
uc	2.50(1.02)	3.10(1.14)	4.30(0.90)	4.80(1.08)	6.00(1.10)	6.90(1.30)	8.70(1.79)	8.00(2.93)
ss	1.10(0.54)	2.20(0.40)	1.70(0.64)	1.50(0.81)	0.60(0.80)	0.40(0.49)	0.10(0.30)	0.00(0.00)
wc	4.10(0.83)	4.30(1.10)	6.70(1.55)	7.50(1.36)	10.1(2.0)	10.6(3.5)	11.6(3.4)	10.9(3.9)
sc	0.90(0.30)	2.20(0.40)	1.70(0.64)	1.60(0.66)	0.90(0.83)	0.40(0.49)	0.20(0.40)	0.00(0.00)
isc	0.90(0.30)	1.30(0.90)	1.30(0.90)	1.40(1.02)	0.60(0.80)	0.30(0.46)	0.10(0.30)	0.00(0.00)
asc	1.10(0.30)	2.90(1.87)	4.90(2.21)	3.10(1.58)	2.50(1.69)	2.10(1.70)	1.60(0.92)	1.80(1.08)

(f) To demonstrate that the search strategies proposed in this paper show good promise we have also done some comparative experiments. We obtained the C-code (**`combo`**) of the algorithm proposed in Martello *et al.* [10] from Pisinger (**`http://www.diku.dk/~pisinger/codes.html`**) and compared it with the proposed algorithm. The experimental results showed (see for instance Fig. 2) that the proposed search strategy is competitive with the best algorithms currently known, especially for the so-called hard instances (strongly correlated, inverse strongly correlated, almost strongly correlated). This must also be evaluated against the background that **`combo`** is the product of more than two decades of research done by the architects of the code, while our code is of experimental nature.

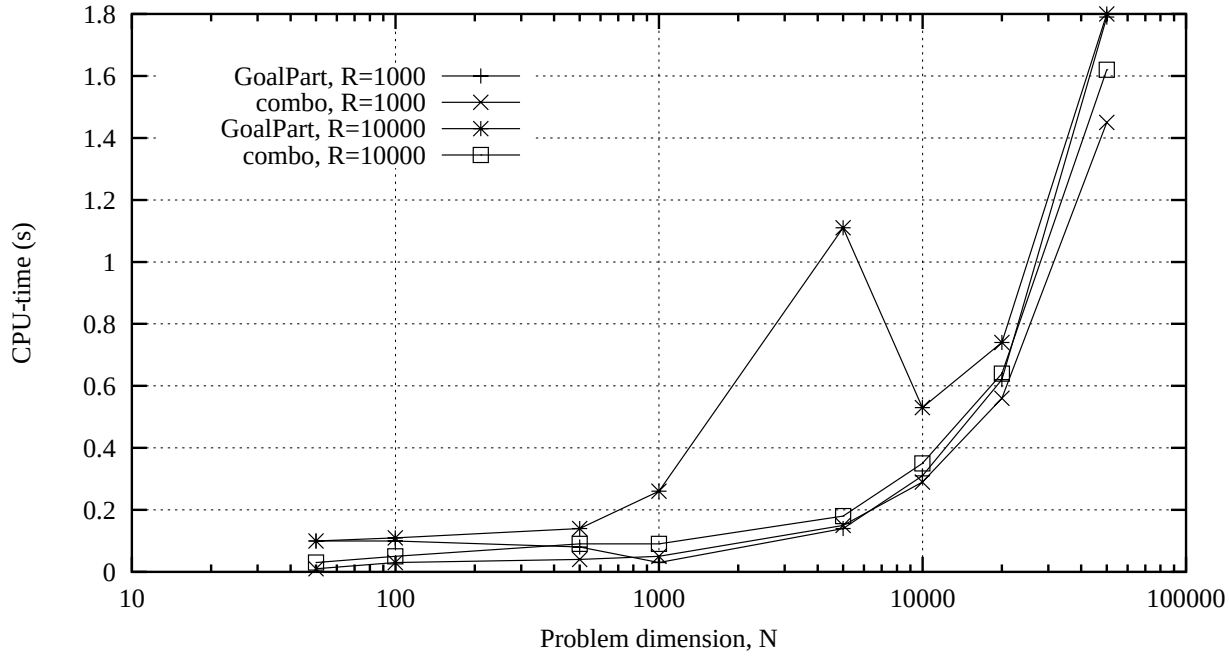

Figure 2: `GoalPart` vs `combo`, Strongly Correlated (sc)

6. CONCLUSION AND FUTURE DIRECTIONS

6.1 Contributions

(a) We have shown that reformulating the 0-1 Knapsack Problem by partitioning the search space through the inclusion of equality constraints on the knapsack's cardinality, can produce very tight upper bounds. To our knowledge, we are the first to introduce *equality* cardinality constraints to explicitly partition the search space. This complements previous investigations on deriving *valid inequalities* to tighten problem relaxations.

(b) The effectiveness of our approach was demonstrated by producing an experimental code which compares very favourably with some of best computer codes currently available. The competitiveness of our approach can mainly be attributed to the pruning power by the inclusion of the cardinality constraints. Another major factor is the efficiency of the algorithms used to solve the Exact k-item Knapsack Problems. This can be attributed to the fact that no searching (in the normal sense) is needed to complete partial solutions containing k - 1 items in the branch-and-bound process. Furthermore, since these completed solutions are integer, the bounds they produce are exact.

(c) Although the number of cardinalities in the cardinality range may be large, we have shown that good feasible solutions found along the way, reduce the computational effort by reducing the number of trees searched and the size of the search trees.

(d) Further investigations into the use of cardinality constraints to partition the search space for general 0-1 Integer Linear Programs with the aim of deriving (novel) high-level parallel algorithms now seem feasible.

6.2 Directions for Future Research

6.2.1 Parallelization

The algorithm proposed in this paper lend itself naturally to a high-level parallel implementation in which each processing node is given a tree to search. We have done some preliminary studies on an IBM SP2 machine with 7 nodes where we have used a master-slave model. To devise a competitive master-slave implementation the following issues need to be addressed to keep the message passing overhead at an acceptable level:

- Is the master also going to search a tree?
- How is the process at each node going to get subproblem data?
    - Is the subproblem data to be written to a file and then read by each process?
    - Is the data going to be packed into a message buffer and passed to the node by some message passing interface like PVM or MPI?
- If a good solution is found by one process, is it to be broadcasted to the other processes?
- If good solutions are to be broadcast to other processes, how often will this be done?

6.2.2 A Partitioning Approach to the Multi-dimensional Knapsack Problems

The LP-solution of the 0-1 Knapsack Problem can be written down analytically as soon as the items have been sorted according to non-decreasing profit densities. We think that the variables of the Multi-dimensional Knapsack Problem formulation can also be sorted according to some criteria used in LP column selection. A partitioning scheme like the proposed one can then be devised which may produce tight bounds. A successful partitioning scheme will certainly have parallelization potential.

ACKNOWLEDGEMENTS

This research was partially funded by the Telkom Grintek *Centre of Excellence* at North-West University.

REFERENCES

[1] E. BALAS and E. ZEMEL, An algorithm for large zero-one knapsack problems, *Operations Research*, 28, 1130-1154 (1980).

[2] R.E. BELLMAN, *Dynamic Programming*, Princeton University Press, Princeton, NJ, (1957).

[3] R.E. BELLMAN and S.E. DREYFUS, *Applied Dynamic Programming*, Princeton University Press, Princeton, NJ, (1962).

[4] G.B. DANTZIG, Discrete variable extremum problems, *Operations Research*, 5, 266-277 (1957).

[5] K. DUDZIŃSKI and S. WALUKIEWICZ, Exact methods for the knapsack problem and its generalizations, *European Journal of Operations Research*, 28, 3-21 (1987).

[6] D. FAYARD and G. PLATEAU, An algorithm for the solution of the 0-1 knapsack problem, *Computing*, 28, 269-287 (1982).

[7] E. HOROWITZ and S. SAHNI, Computing partitions with applications to the knapsack problem, *Journal of ACM*, 21, 277-292 (1974).

[8] P.J. KOLESAR, A branch and bound algorithm for the knapsack problem, *Management Science*, 13, 723-735 (1967).

[9] M.F. KRUGER, *State space search models for discrete optimization, Knapsack algorithms and related problems*, PhD thesis, School of Computer, Statistical and Mathematical Sciences, Potchefstroom University for Christian Higher Education, Potchefstroom, South Africa (1998).

[10] S. MARTELLO, D. PISINGER and P. TOTH, Dynamic programming and tight bounds for the 0-1 knapsack problem, *Management Science*, 45, 414-424 (1999).

[11] S. MARTELLO and P. TOTH, An upper bound for the zero-one knapsack problem and a branch and bound algorithm, *European Journal of Operations Research*, 1, 169-175 (1977).

[12] S. MARTELLO and P. TOTH, *Knapsack Problems*: *Algorithms and Computer Implementations*, Wiley, Chichester, England (1990).

[13] S. MARTELLO and P. TOTH, Upper bounds and algorithms for hard 0-1 knapsack problems, *Operations Research*, 45, 768-783 (1997).

[14] R.M. NAUSS, An efficient algorithm for the 0-1 knapsack problem, *Management Science*, 23, 27-31 (1976).

[15] S.N.N. PANDIT and M. RAVI KUMAR, A lexicographic search for strongly correlated 0-1 knapsack problems, *Opsearch*, 30, 76-116 (1993).

[16] D. PISINGER, *Algorithms for Knapsack Problems*, PhD thesis, DIKU, University of Copenhagen, Denmark (1995).

[17] P. TOTH, Dynamic programming algorithms for the zero-one knapsack problem, *Computing*, 25, 29-45 (1980).