

ORDERED ENUMERATION METHOD

Felix Friedman
Department of Computer Science
East Stroudsburg University
East Stroudsburg, PA 18301

ABSTRACT

Integer programming algorithms based on the ordered enumeration method are described. Combining dynamic programming and branch and bound ideas in one efficient computational process, S. S. Lebedev implemented his method for solving integer linear programming problems in 1968. A number of ordered enumeration algorithms were developed since the 1970's. The author has been actively involved in this research and the paper presents several such algorithms, extended application and parallelization of the method.

Most papers on this highly competitive combinatorial method of discrete optimization are written in Russian and little known to researchers and educators not familiar with the language.

Key Words. combinatorial method, branch and bound method, dynamic programming, ordered enumeration method, integer programming

ORDERED ENUMERATION METHOD

GENERAL CONCEPT

Let $f(x)$ and $F(x)$ be two functions defined on finite sets S and T respectively. Given $S \subset T \subset \mathbb{R}^n$ and $f(x) \leq F(x)$ for $x \in S$, consider a problem

$$\max_{x \in S} f(x) \quad (1)$$

Assume an algorithm, generating T in an order of nonincreasing values of $F(x)$, has already produced a sequence of x^1, x^2, \dots, x^t , $t \geq 1$ elements from T and $S^* = S \cap \{x^1, x^2, \dots, x^t\}$.

Lemma

If S^* is not empty and for some index h , $1 \leq h \leq t$, $F(x^h) \leq f(x^h)$, where

$$f(x^h) = \max_{x \in S^*} f(x)$$

then x^h is an optimal solution of (1).

Proof

If $1 \leq p \leq t$ and $x^p \in S$ then $f(x^h) \geq f(x^p)$, while for any other generated element $x^p \in T$, $p > t$, such that $x^p \in S$, $f(x^p) \leq F(x^p) \leq F(x^t) \leq f(x^t)$. Therefore, $f(x^h) \geq f(x^p)$ for any $x^p \in S$.

ORDERED ENUMERATION METHOD FOR A MULTIDIMENSIONAL 0/1 KNAPSACK PROBLEM [15]

Consider an m -dimensional Knapsack problem represented in form

$$z_{\text{opt}} = \max_{x \in S} \sum_{j=1}^n c_j x_j \quad (2)$$

where

$S = \{x = (x_1, x_2, \dots, x_n) \mid \sum_{j=1}^n a_{ij} x_j \leq b_i, x_j \in \{0, 1\}, i = \overline{1, m}; j = \overline{1, n}\}$ and a_{ij}, b_i are nonnegative, c_j are positive integers.

Consider also a vector

$$I = (I_1, I_2, \dots, I_m), I_i \geq 0, i = \overline{1, m} \text{ and let } a_j = \sum_{i=1}^m a_{ij} I_i, j = \overline{1, n}; b = \sum_{i=1}^m b_i I_i;$$

$$T = \{x = (x_1, x_2, \dots, x_n) \mid \sum_{j=1}^n a_j x_j \leq b, x_j \in \{0, 1\}, j = \overline{1, n}\}; F(x) = f(x) = \sum_{j=1}^n c_j x_j.$$

The choice of vector I in producing a so-called surrogate knapsack constraint, defining set T, is an important issue and is addressed later.

Assuming an existence of an algorithm generating T in nonincreasing order of $\sum_{j=1}^n c_j x_j$, the ordered enumeration method is applied for solving an m-dimensional Knapsack problem. Notice that $S \subset T$ and

$$\bar{z} = \max_{x \in T} \sum_{j=1}^n c_j x_j \quad (3)$$

is a unidimensional Knapsack problem with $\bar{z} \geq z_{opt}$.

To solve the m-dimensional Knapsack problem, all solutions from T with $F(x) = \sum_{j=1}^n c_j x_j = \bar{z}$ are generated

first, then all solutions from T with $F(x) = \sum_{j=1}^n c_j x_j = \bar{z} - 1$ and so on. The process is terminated when a generated solution from T belongs also to S, in other words, satisfies all m-constraints. According to lemma above this solution is optimal. $\sum_{j=1}^n c_j x_j$ may assume integer values only from the set $\{\bar{z}, \bar{z} - 1, \bar{z} - 2, \dots, 0\}$ since each c_j is a positive integer.

ALGORITHM FOR GENERATING T

A dynamic programming algorithm (see, for example, [18], p. 699) determines optimal solution(s) of (3). An extension of it, introduced under the name Ordered Enumeration Algorithm for a Knapsack Problem by S. S. Lebedev in 1968 [15], generates all solutions (x_1, x_2, \dots, x_n) from T in an order of nonincreasing values of

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

The recurrence relations for this algorithm are obtained in the following way [6]. Let $\{g_k(w)\}$ be the set of values

$$z = \sum_{j=1}^k c_j x_j \text{ on a set } \{(x_1, x_2, \dots, x_k) \mid \sum_{j=1}^k a_j x_j \leq w, x_j \in \{0, 1\}, j = \overline{1, k}\}, w \leq b, \text{ and}$$

$$\{g_k(w)\} = \{-\infty\}, k = \overline{0, n}, \text{ for } w < 0; g_0(w) = \{0\} \text{ for } w \geq 0.$$

$$\text{Then } \{g_k(w)\} = \{g_{k-1}(w)\} \cup \{g_{k-1}(w - a_k) + c_k\}, k = \overline{1, n}. \quad (4)$$

It may be noticed that if $z \in \{g_k(w_1)\}$ then $z \in \{g_i(w)\}, i \geq k, w \geq w_1$.

A family of sets $\{g_k(w)\}, k = 1, 2, \dots, n$, can be represented by triples $[z_i, j, w_i], j \leq k$, where w_i is determined from conditions $z_i \in \{g_j(w)\}$ if $w \geq w_i$ and $z_i \notin \{g_j(w)\}$ if $w < w_i$.

It is said that $[z_1, j_1, w_1]$ dominates $[z_2, j_2, w_2]$ if $j_1 \leq j_2, w_1 \leq w_2, z_1 \geq z_2$.

Triples with the same z can be grouped in a family. Only a triple which is not dominated by any already existing triple in the family joins it while computations are performed according with formula (4). By comparison, only a triple which is not dominated by any existing triple from any family joins the corresponding family in the dynamic programming algorithm. The ordered enumeration algorithm is comprised of PART 1 and PART 2. PART 1 generates a table of not dominated triples representing sets $\{g_0(w)\}, \{g_1(w)\}, \dots, \{g_n(w)\}, w \leq b$.

PART 2 of the algorithm generates solutions from T (i.e., solutions satisfying the Knapsack constraint) in an order

of nonincreasing values of $\sum_{j=1}^n c_j x_j$, starting with the value \bar{z} , followed by $\bar{z} - 1, \bar{z} - 2, \dots, 0$. A solution $(x_1, x_2,$

..., x_n) for a value z_0 is constructed in n sequential steps by testing whether 0 or 1 can be assigned to $x_k, k=n, n-1, \dots, 1$.

Let $z^{(k-1)} = z_0 - \sum_{j=k}^n c_j x_j$ and $w^{(k-1)} = b - \sum_{j=k}^n a_j x_j$. When testing $x_k = \bar{x}_k$, where \bar{x}_k is either 0 or 1, the

algorithm investigates whether $z^{(k-1)} \in \{g_{k-1}(w^{(k-1)})\}$. If it is true, the assignment is possible. First the 0-assignment is tested. If it is not possible, then $x_k = 1$. If 0-assignment is possible, then $x_k = 0$ and 1-assignment is tested. If it is also possible, it is remembered by storing index k in a list (stack) L , and the process is repeated for $j=k-1 > 0$.

ORDERED ENUMERATION ALGORITHM FOR UNIDIMENSIONAL 0/1 KNAPSACK PROBLEM ([15])

PART I

1. $k := 0$. Store $[0, 0, 0]$.
2. $k := k + 1$. For each z locate a stored triple $[z, j, w], j < k$, with the smallest w and compute a triple $[z + c_k, k, w + a_k]$; store it, if $(w + a_k \leq b)$ and (there is no stored triple $[z', j, w']$ with $z' = z + c_k, j < k, w' \leq w + a_k$).
3. If $k = n$ then go to PART 2 else go to 2.

PART 2

Let $z^{(1)} = \bar{z}$, $z^{(2)} = \bar{z} - 1$, $z^{(3)} = \bar{z} - 2$, ..., $z^{(s)} = 0, (s = \bar{z} + 1)$. Let L be a list of indices where a branching for a current solution is recorded and removed in a prescribed way.

1. $p := 1, L := \emptyset$.
2. $j := n, z_0 := z^{(p)}, w_0 := b$.
3. If there is a triple $[z_0, k, w]$ with $k < j, w \leq w_0$ then $x_j := 0$ else go to 5.
4. If there is a triple $[z_j - c_j, k, w]$ with $k < j, w \leq w_0 - a_j$ then record j in L . Go to 6.
5. $x_j := 1, z_0 := z_0 - c_j, w_0 := w_0 - a_j$.
6. If $j = 1$ then a solution (x_1, x_2, \dots, x_n) with $z^{(p)} = \sum_{j=1}^n c_j x_j$ is generated else $j := j - 1$ and go to 3.
7. If $L \neq \emptyset$ then (remove j -the last recorded index from $L, x_j := 1, z_0 := z^{(p)} - \sum_{k=j}^n c_k x_k, w_0 := b - \sum_{k=j}^n a_k x_k$) and go to 6.
8. If $(p \neq s)$ and $(L = \emptyset)$ then $p := p + 1$ and go to 2.
9. Stop.

NUMERICAL EXAMPLE

Consider a Knapsack Problem

$$\max z = 4x_1 + 5x_2 + x_3 + 3x_4 + x_5$$

subject to

$$x_1 + 3x_2 + x_3 + 4x_4 + 2x_5 \leq 5, x_j \in \{0, 1\}, j = \overline{1, 5}.$$

PART 1

Triples $[k, w, z]$ with the same z are presented as tuples $[k, w]$ in column z :

z	0	1	2	3	4	5	6	7	9	10
$[k, w]$	$[0, 0]$	$[3, 1]$	$[5, 3]$	$[4, 4]$	$[1, 1]$	$[2, 3]$	$[3, 4]$	$[4, 5]$	$[2, 4]$	$[3, 5]$
						$[3, 2]$				

PART 2

Assume $T(10)$, the set of solutions with $z=10$, is to be determined. Assignment $x_5=1$ is impossible: $z(4)=10-c_5=9$, $w(4)=5-a_5=3$, while the only tuple in column 9 has $w=4 > 3$. Therefore, $x_5=0$. Similarly, assignment $x_4=1$ is also impossible and $x_4=0$. Assignment $x_3=1$ is possible ($z(2)=10-1=9$, $w(2)=5-1=4$ and column 9 has a tuple with $w \leq w(2)=4$). Assignment $x_3=0$ is impossible. The assignments $x_2 = x_1 = 1$ are only possible ones and $T(10) = \{(1, 1, 1, 0, 0)\}$.

ORDERED ENUMERATION ALGORITHM FOR MULTIDIMENSIONAL 0/1 KNAPSACK PROBLEM ([15], [7])

A multidimensional 0/1 knapsack problem may be solved by finding a special solution of a set of unidimensional knapsack problems each having one different constraint and the same objective function of the initial problem. This special solution satisfies each constraint and delivers a value of the objective function which is not less than any other solution of the set.

Given an m-dimensional Knapsack problem (2), S is the set of its feasible solutions and T is a set of feasible solutions of a Knapsack problem (3). To insure that $T \supset S$, the Knapsack constraint is computed using a vector $\bar{I} = (\bar{I}_1, \bar{I}_2, \dots, \bar{I}_m)$, $\bar{I}_i \geq 0$, $i = 1, \dots, m$. Different choices of \bar{I} affect the optimal value of the resulting

Knapsack problem \bar{z} , which is used as an upper bound for z_{opt} in (2) [16]. Straightforward implementation of the ordered enumeration method is achieved by employing the ordered enumeration algorithm for a Knapsack problem to generate its solutions starting with solutions yielding \bar{z} and continuing with solutions yielding decreasing values

for $\sum_{j=1}^n c_j x_j$. Since $f(x)=F(x)=\sum_{j=1}^n c_j x_j$ the first solution generated in this process which satisfies all m constraints of (2) is an optimal solution of (2). A more efficient implementation of the ordered enumeration algorithm is achieved by considering more than one Knapsack problem of type (3) when solving (2). Let a feasible

solution of (3) for some $z_0 = \sum_{j=1}^n c_j x_j$, $z_{opt} \leq z_0 \leq \bar{z}$, is being constructed and $n-k-1 \geq 0$ components have

been already fixed: $x_n = \bar{x}_n, x_{n-1} = \bar{x}_{n-1}, \dots, x_{k+2} = \bar{x}_{k+2}; k \leq n-1$ and $x_{k+1} = \bar{x}_{k+1}$ is being tested.

Let $z(k)=z_0 - \sum_{j=k+1}^n c_j \bar{x}_j, w_i(k)=b_i - \sum_{j=k+1}^n a_{ij} \bar{x}_j, i=1, \dots, m$.

The assignment $x_{k+1} = \bar{x}_{k+1}$ is possible if existence of (x_1, x_2, \dots, x_k) , such that $\sum_{j=1}^k c_j x_j = z(k)$ and

$\sum_{j=1}^k a_{ij} x_j \leq w_i(k), x_j \in \{0, 1\}, j=1, \dots, k$, is established for each i independently.

Notice that $(\bar{x}_1, \bar{x}_2, \dots, \bar{x}_k)$ is not necessarily the same for different values of i since $\sum_{j=1}^k a_{ij} \bar{x}_j \leq w_i(k)$ and

$\sum_{j=1}^k c_j x_j = z(k)$ could be satisfied by more than one such vector.

If such (x_1, x_2, \dots, x_k) does not exist for $i=i_0$ the other assignment of x_{k+1} is tried. If either assignment (0 or 1) is impossible a solution with $x_n = \bar{x}_n, x_{n-1} = \bar{x}_{n-1}, \dots, x_{k+2} = \bar{x}_{k+2}$ is not feasible for (2).

In this case we name $(\bar{x}_n, \bar{x}_{n-1}, \dots, \bar{x}_{k+2})$ an incomplete variant. This discussion leads to an implementation with several unidimensional Knapsack problems with the same objective function solved simultaneously. Constraints

$\sum_{j=1}^n a_{ij} x_j \leq b_i$ from (2) may be used as a source of Knapsack constraints for these problems. The algorithm

below solves the problem (2) with additional commonly encountered (in applications) "ladder-like" constraints

$$\sum_{j=r(p)}^{h(p)} x_j \leq 1, p = 1, 2, \dots, q, \quad (5)$$

where $r(1)=1, r(p) \leq h(p), p=1, 2, \dots, q; h(1) \leq h(2) \leq \dots \leq h(q)=n$.

The constraints (5) are represented by a vector $(s(1), s(2), \dots, s(n))$, where $s(j)$ is the maximal index k from a segment of successive integers $[0, \dots, j-1]$ such that both x_k and x_j are not in the same constraint from (5). The recurrence relations for this algorithm may be received by modifying the respective relations to

$g_k(w) = \{\infty\}, w < 0, k=0, 1, 2, \dots, n; g_0(w) = \{0\}, w \geq 0; \{g_k(w)\} = \{g_{k-1}(w)\} \cup \{g_{s(k)}(w-a_k)+c_k\}, k=1, 2, \dots, n$.

Constraints (5) allow to reduce the linear integer programming problem to an equivalent problem with boolean variables, thus, making the method more general.

The problem (2) can be considered as a special case of the one under discussion with $s(j) = j-1, j=1, \dots, n$. The following algorithm for the problem utilized m_1 first Knapsack constraints of (1). The PART 1 generates m_1 tables of triples, one for each unidimensional Knapsack problem

$$\max \sum_{j=1}^n c_j x_j \mid \left(\sum_{j=1}^n a_{ij} x_j \leq b_i, x_j \in \{0,1\}, j = \overline{1, n} \right), i = 1, \dots, m_1.$$

The value of m_1 may vary from 1 to m , increasing m_1 results usually in a smaller \bar{z} and shorter incomplete variants thus reducing the solution time. In the original implementation [15] the subset of triples $\{[z, k, w]\}$ is computed after the subset of triples $\{[z', j, w'] \mid j < k\}$ is already computed. If a part of the second subset is stored in secondary memory (which is a valid assumption for large n), then the frequent transfer of triples to main memory (which could not be avoided in this case) increases the solution time considerably. The implementation [7] makes the use of secondary memory (when it is necessary) more efficient. Notice that triples with fixed z_1 , are generated from triples with $z = z_1 - c_j, j = \overline{1, n}$, only. The new implementation [7] creates triples for $z_1 = 0, 1, 2, \dots, \bar{z}$ successively which allows to keep triples with $z < z_1 - \max_j c_j$ in secondary memory while generating triples with z_1 .

The triple $[0,0,0]$ is stored first. Assuming all triples with $z \leq z_1 - 1$ are already stored, the triples for z_1 are generated in following way. Sequentially, for each $k=1, 2, \dots, n$ a subset of triples $Q_k = \{[z_1 - c_k, j, w] \mid j < k\}$ is considered and a triple $[z_1 - c_k, j', w']$ is located with minimal w' , a triple $[z_1, k, w_1]$, where $w_1 = w' + a_k$ is stored if $w_1 \leq b$ and no triple $[z_1, j, w]$ with $w \leq w_1$, is stored earlier. In the backtracking part (PART 2 of the algorithm) main memory must keep

triples with $z = z(k) = z_0 - \sum_{j=k+1}^n c_j x_j$ when the assignment $x_{k+1} = \bar{x}_{k+1}$ is tested. If they are not there, they are

transferred from secondary memory. To avoid frequent transfers, triples with $z = z(k-1), z(k-2), \dots, z(k-j_k)$ are transferred also. The size of the integer j_k is chosen as large as the size of main memory will permit. Triples can be stored in a table with $\bar{z} + 1$ columns, corresponding to $z = 0, 1, 2, \dots, \bar{z}$. A triple $[z, j, w]$ is stored in column z of the table as a tuple $[j, w]$. A separate table is maintained for each of m_1 unidimensional Knapsack problems.

ALGORIHM 1 ([7])

PART 1

1. Store tuple $[0,0]$ in column 0 of each of m_1 tables, $\bar{z} := 0, z := 0$.
2. $z := z + 1$. If $z - \bar{z} > \max_j c_j$ then go to PART 2.
3. $i := 0, j := 0$.
4. $j := j + 1$. If $j > n$ then go to 2.
5. $z_1 := z - c_j$. If no tuples in column z_1 then go to 4.
6. $i := i + 1$. If $i > m_1$ then go to 10 else $r := p$, where p is the position in the column z_1 of table i ($p = 1, 2, 3, \dots$) of the tuple stored most recently. The first stored tuple in column z_1 gets the position 1, second gets position 2 and so on.
7. Let $[k_r^{(i)}(z_1), w_r^{(i)}(z_1)]$ be a tuple for z stored in position r in table i . If $w_i = w_r^{(i)}(z_1) + a_j \leq b_i$ then go to 8 else goto 4.
8. If $k_r^{(i)}(z_1) \leq s(j)$ then keep $w_i(z) = w_i$ in memory and go to 6 else go to 9.
9. $r := r - 1$. If $r = 0$ then go to 4 else go to 7.
10. Consider the last tuple for z in table $i, i = 1, 2, \dots, m_1$. If tuple $[k^{(i)}(z), w^{(i)}(z)]$ is such that $w_i(z) < w^{(i)}(z)$ or if the z -column is empty then record tuple $[j, w_i(z)]$ for z in table i . If z did not have any tuples before then $\bar{z} := z$. Go to 4.

PART 2

1. $z := \bar{z} + 1, L := \emptyset$.
2. $j := n + 1$.
3. $w_i := b_i, i = 1, 2, \dots, m; z := z - 1$. If $z > 0$ go to 4 else record $x_1 = x_2 = \dots = x_n = 0$ and stop.
4. $j := j - 1$. If $j > 0$ then go to 5 else an optimal solution is received. Display it and go to 9.

5. Establish whether z has a tuple $[k^{(i)}(z), w^{(i)}(z)]$ in table i , such that $k^{(i)}(z) < j$ and $w^{(i)}(z) \leq w_i$. If such tuple exists for each i from $\{1, \dots, m_i\}$ then $x_j^0 := 0$ else $x_j^0 := -1$.
6. $z_1 := z - c_j$, $w_1 := w_i - a_{ij}$, $i=1, \dots, m$. If any of $m+1$ numbers $z_1, w_1^{(1)}, w_1^{(2)}, \dots, w_1^{(m)}$ is negative go to 8.
7. Establish whether z_1 has a tuple $[k^{(i)}(z_1), w^{(i)}(z_1)]$ in table i , such that $k^{(i)}(z_1) \leq s(j)$ and $w^{(i)}(z_1) \leq w_1(i)$. If such tuple exists for each i from $\{1, 2, \dots, m_i\}$ then $x_j^1 := 1$ else $x_j^1 := -1$.
8. If $x_j^0 = x_j^1 = -1$ then go to 9. If $x_j^0 = 0$ then $x_j := 0$. If $x_j^0 = 0$ and $x_j^1 = 1$, record index j in list L . If $x_j^1 = 1$ and $x_j^0 = -1$ then $x_j := 1$, $w_i := w_1^{(i)}$ $i = 1, 2, \dots, m$. Go to 4.
9. If $L = \emptyset$ and an optimal solution is not yet received then go to 2. If $L = \emptyset$ and an optimal solution is received then stop. If $L \neq \emptyset$ and \bar{j} is the last recorded index in L then remove \bar{j} from L , $j := s(\bar{j}) + 1$, $x_j := 1$, $x_{\bar{j}-1} := 0, \dots, x_{\bar{j}} := 0$;

$$z := z - \sum_{k=j}^n c_k x_k, w_i = b_i - \sum_{k=j}^n a_{ik} x_k, i = \overline{1, m}; \text{ and go to 4.}$$

MODIFIED ALGORITHM ([8], [7])

(Algorithm 2)

Some incomplete variants have a tendency to be generated in PART 2 of the algorithm again and again while solutions for $z = \bar{z}, \bar{z} - 1, \bar{z} - 2, \dots$ are being constructed. Repeated variants $(\bar{x}_n, \bar{x}_{n-1}, \dots, \bar{x}_k)$ are due to the fact that $z = \bar{z}, \bar{z} - 1, \bar{z} - 2, \dots$ could be received with $x_j = \bar{x}_j, j \geq k$, and various assignments of $x_j, j < k$, none of which satisfies all Knapsack constraints in (2) - this becomes known after the assignment of $x_k = \bar{x}_k$. Let $T(z)$ be the set of variants generated in PART 2 of algorithm 1 when constructing solutions for z . PART 2 generates sets $T(\bar{z}), T(\bar{z} - 1), T(\bar{z} - 2), \dots$ successively. This process stops with generating $T(\bar{z} - p)$ when the first time a set $T(z)$ gets a completed variant which is a feasible solution and, therefore, an optimal solution of (2). This defines $z_{opt} = \bar{z} - p$. When a variant from $T(z)$ is under construction 0 and 1 values of x_k are tested. If x_k can be equal to 0 then it is checked whether x_k can be 1. If two values of x_k are possible then 0 is assigned to x_k for the current variant and k is recorded in a list L to be able to continue the current variant with $x_k = 1$ later. In the algorithms above, indices, where a variant has another branch, are remembered for current z_0 only. In a new approach [8] a current variant is checked whether it belongs to

$T(z)$ for $z = z_0 - 1, z_0 - 2, \dots, \hat{z}$, where \hat{z} is a value of $\sum_{j=1}^n c_j x_j$ corresponding to some feasible solution found by

an approximation method. Branches from the current variant are remembered for all $z, \hat{z} \leq z \leq z_0$, and when coming to a new z the algorithm starts from incomplete variants remembered for this z . Let $M_{n+1} = \{\hat{z}, \hat{z} + 1, \dots, \bar{z}\}$ and

$(x_n, \dots, x_{k+1}), k < n$, be the current variant from $T(z_0)$, $\hat{z} \leq z_0 \leq \bar{z}$ and

$w_i = b_i - \sum_{j=k+1}^n a_{ij} x_j, i = \overline{1, m}$, $z = z_0 - \sum_{j=k+1}^n c_j x_j$ are already computed. Assume also that a set M_{k+1} of such

z_t for which (x_n, \dots, x_{k+1}) is a variant also is generated. If $x_k = 0$ is possible then M_k for the variant $(x_n, \dots, x_{k+1}, x_k = 0)$ can be found by excluding those z_t from M_{k+1} , for which $x_k = 0$ is impossible. If $x_k = 0$ is the only possible assignment for $z = z_0$ then the largest z_{t_0} is considered among those z_t for which $x_k = 1$ is a possible assignment and variant $(x_n, x_{n-1}, \dots, x_{k+1}, x_k = 1)$ is recorded in a special table $P(z_{t_0})$. This variant does not have to be remembered for $z_t < z_{t_0}$, $z_t \in M_{k+1}$

since it is to be considered when generating variants for z_{t_0} . If for $z = z_0$, $x_k = 0$ and $x_k = 1$ are possible, then index k is recorded in list L . Branches for $z_t \leq z_0, z_t \in M_{k+1}$, do not have to be remembered since they will be considered when a corresponding branch is constructed for $T(z_0)$. If for $z = z_0$ the only possible assignment is $x_k = 1$, then M_k is found for $(x_n, x_{n-1}, \dots, x_{k+1}, x_k = 1)$ by excluding those z_t from M_{k+1} for which $x_k = 1$ is impossible. The incomplete variant $(x_n, x_{n-1}, \dots, x_{k+1}, x_k = 0)$ is recorded in a special table $P(z_{t_0})$. This variant does not have to be remembered for $z_t < z_{t_0}$, $z_t \in M_{k+1}$ since it is to be considered when generating variants for z_{t_0} . If for $z = z_0$, $x_k = 0$ and $x_k = 1$ are possible, then index k is recorded in list L . Branches for $z_t \leq z_0, z_t \in M_{k+1}$, do not have to be remembered since they will be considered when a corresponding branch is constructed for $T(z_0)$. If for $z = z_0$ the only possible assignment is $x_k = 1$, then M_k is found for $(x_n, x_{n-1}, \dots, x_{k+1}, x_k = 1)$ by excluding those z_t from M_{k+1} for which $x_k = 1$ is impossible. The incomplete variant $(x_n, x_{n-1}, \dots, x_{k+1}, x_k = 0)$ is recorded in a special table $P(z_{t_0})$. This variant does not have to be remembered for $z_t < z_{t_0}$, $z_t \in M_{k+1}$ since it is to be considered when generating variants for z_{t_0} .

$1, \dots, x_{k+1}, 0$) is recorded in a table $P(z_{t_0})$ where z_{t_0} is the largest among $z_i \in M_{k+1}$, for which $x_k=0$ is impossible. At last it can happen that neither $x_k=0$ nor $x_k=1$ is possible for $z=z_0$. It means that the current variant can not be continued for $z=z_0$. It might be continued for other $z \in M_{k+1}$. Therefore, the largest $z_{t_1} \in M_{k+1}$ with a possible assignment $x_k=0$ and the largest $z_{t_2} \in M_{k+1}$ with a possible assignment $x_k=1$ are found. Incomplete variants $(x_n, x_{n-1}, \dots, x_{k+1}, 0)$ and $(x_n, x_{n-1}, \dots, x_{k+1}, 1)$ are recorded in tables $P(z_{t_1})$ and $P(z_{t_2})$ respectively. (It is possible $z_{t_1} = z_{t_2}$). If L is not empty and s is the last in L then s is removed from L , a variant $(x_n, x_{n-1}, \dots, x_{s+1}, x_s=1)$ is created and $z=z_0 - \sum_{j=s}^n c_j x_j, w_i=b_i - \sum_{j=s}^n a_{ij} x_j, i=\overline{1, m}$, are computed, M_s is defined as a set $\{\hat{z}, \hat{z}+1, \dots, z_0-1\}$ and construction of this variant is continued. If L is empty and $\hat{z} \leq z_0 < \bar{z}$ then an incomplete variant is removed from $P(z_0)$ and processed in a similar way. If (L is empty and $z_0=z$) or ($L = \emptyset$ and $P(z_0) = \emptyset$) then $z_0 := z_0 - 1$ and incomplete variants from $P(z_0)$ are considered. Notice that the tables $P(z)$ may be kept in secondary memory. If $z_0 < \bar{z}$ then $P(z_0)$ is transferred to main memory.

NUMERICAL EXAMPLE

Maximize

$$2x_1 + 3x_2 + 2x_3 + 4x_4 + 4x_5 + 3x_6 + 3x_7 + 4x_8 \leq 5$$

Subject to

$$\begin{array}{rcccccccc} 2x_1 & + & 2x_2 & + & x_3 & + & 2x_4 & + & 2x_5 & + & 2x_6 & + & 2x_7 & + & x_8 & \leq & 5 \\ x_1 & + & x_2 & & & + & x_4 & + & x_5 & & & + & x_7 & & & \leq & 3 \\ x_1 & & & + & x_3 & & + & x_5 & + & x_6 & & & + & x_8 & & \leq & 1 \\ & & x_2 & & & + & x_4 & & & + & x_6 & + & x_7 & & & \leq & 1 \\ x_1 & + & x_2 & + & x_3 & & & & & & & & & & & \leq & 1 \\ & & x_2 & + & x_3 & + & x_4 & & & & & & & & & \leq & 1 \\ & & & & & & & x_5 & + & x_6 & & & & & & \leq & 1 \\ & & & & & & & & & x_6 & + & x_7 & & & & \leq & 1 \\ & & & & & & & & & & & & & & & x_8 & \leq & 1 \end{array}$$

$$x_j \in \{0, 1\}, j = \overline{1, 8}$$

z	0	1	2	3	4	5	6	7	8	9	10	11	i
[0, 0]		[1, 2]	[2, 2]	[4, 2]	[7, 3]	[4, 4]	[5, 4]	[5, 4]	[7, 5]	[8, 4]	[8, 5]		1
		[3, 1]		[8, 1]		[5, 3]	[8, 3]	[8, 3]					
[0, 0]		[1, 1]	[2, 1]	[4, 1]	[7, 1]	[4, 2]	[5, 2]	[5, 2]	[7, 2]	[8, 1]	[8, 1]		2
		[3, 0]	[6, 0]	[8, 0]		[5, 1]	[6, 1]	[8, 1]					
						[8, 0]							
[0, 0]		[1, 1]	[2, 0]	[4, 0]	[7, 1]	[4, 1]	[5, 1]	[5, 1]	[7, 1]	[8, 1]	[8, 1]		3
						[7, 0]	[7, 0]	[8, 1]					
[0, 0]		[1, 0]	[2, 1]	[4, 1]	[7, 1]	[4, 1]	[5, 1]	[5, 1]	[7, 1]	[8, 0]	[8, 1]		4
			[5, 0]		[5, 0]		[8, 0]						

The tables define $\bar{z}=11$. This is largest value of z marked by tuples in the tables above. Applying PART 2 of the Algorithm 2, the optimal solution(s) can be found. Notice, that $x_1=x_2=x_3=0, x_4=x_5=1, x_6=x_7=x_8=0$ is a feasible solution and delivers 8 to the objective function. Assuming $\hat{z}=8$ we may invoke the PART 2 and receive two optimal solutions $x_1=x_2=x_3=x_5=x_6=x_7=0, x_4=x_8=1$ and $x_1=x_2=x_3=x_6=x_7=x_8=0, x_4=x_5=1$.

EXTENDED APPLICATIONS ([5], [11])

Algorithms 1 and 2 are presented as algorithms for linear problems with boolean variables and ladder-like additional constraints. However, they have a larger scope of applications. Following are examples of them.

(a) A linear integer programming problem

$$\max \left\{ \sum_{j=1}^n c_j x_j \mid \sum_{j=1}^n a_{ij} x_j \leq b_i, i = \overline{1, m}; x_j \in \{0, 1, 2, \dots, d_j\}, j = \overline{1, n} \right\}$$

(b) a separable integer programming problem

$$\max \left\{ \sum_{j=1}^n f_j^o(x_j) \mid f_j^i(x_j) \leq b_i, i = \overline{1, m}; x_j \in \{0, 1, 2, \dots, d_j\}, j = \overline{1, n} \right\} \text{ can be solved by substitution}$$

$$x_j = \sum_{k=1}^{d_j} k x_{jk}, \quad \sum_{k=1}^{d_j} x_{jk} \leq 1, \quad x_{jk} \in \{0, 1\}, j = \overline{1, n} \text{ and representation}$$

$$f_j^i(x_j) = \sum_{k=1}^{d_j} f_j^i(k) x_{jk}, \quad i = \overline{0, m}; j = \overline{1, n}; \text{ functions } f_j^i(x_j), i = \overline{0, m}, \text{ are assumed nonnegative and}$$

$$f_j^i(0) = 0, i = \overline{1, m}, j = \overline{1, n}.$$

(c) The commonly encountered multiple-choice Knapsack problem

(see, for example, [19]) is a special case of a Knapsack problem with additional ladder-like constraints.

(d) Linear integer programming problems with additional special

constraints and logical conditions. The following problem is an example:

$$\sum_{i=1}^m \sum_{p=0}^{T-1} \sum_{p < t \leq T} c_{it}^p x_{it}^p \rightarrow \max$$

subject to

$$\sum_{t \geq s} \sum_{0 \leq p < s} x_{it}^p \leq 1, s = 1, \dots, T; i = 1, \dots, m;$$

$$\sum_{p=0}^{T-1} \sum_{t > p} a_{it}^p x_{it}^p \leq b_i, i = 1, \dots, m; x_{it}^p \in \{0, 1\}, t = 1, \dots, T; p = 0, \dots, T-1, p < t; i = 1, 2, \dots, m$$

and logical conditions

$$\text{if for } p \in \{1, \dots, T-1\} \text{ and } i \in \{1, 2, \dots, m\} \sum_{t > p} x_{it}^p = 1 \text{ then } \sum_{g < p} x_{ig}^g = 1.$$

Parameters c_{it} , a_{it} , b_i are assumed positive. It is a problem of an economic-based system in which a number of limited resources are to be allocated with an optimal strategy over several different projects in a multistage development. This approach may be applied to an application with stagewise build-up or down-scaling.

An ordered enumeration algorithm was developed to solve this problem [9]. The logical conditions are satisfied by adjusting the logic of the algorithm.

(e) Special nonlinear integer programming problem

$$\max \left\{ \sum_{j=1}^n c_j x_j - h(x_1, x_2, \dots, x_n) \mid \sum_{j=1}^n a_{ij} x_j \leq b_i, i = \overline{1, m}; x_j \in \{0, 1\}, j = \overline{1, n} \right\},$$

$c_j > 0, a_{ij} \geq 0, b_i \geq 0, j = \overline{1, n}; i = \overline{1, m}$ and $h(x_1, x_2, \dots, x_n)$ is a nonnegative function defined on $S \subset R^n$. If

$$\sum_{j=1}^n c_j x_j \text{ is considerably larger than } h(x_1, x_2, \dots, x_n) \text{ then the optimal solution can be found in a}$$

reasonable amount of time by applying the general concept of the ordered enumeration method for solutions generated by algorithm 1 or algorithm 2.

A SMALLER UPPER BOUND ([17], [1], [2])

Problems (2) and (3) belong to the class NP-Hard problems. An efficiency study of algorithms 1 and 2 [10] showed that the amount of computation in PART 2 is often in many times larger than in PART 1. The processing time of PART 1 is increasing when m_i is increasing. Although the amount of triples in a separate table is decreasing the total amount grows considerably. The decrease of solution time observed for $m_i > 1$ in comparison with $m_i = 1$ is due to smaller \bar{z} and the amount and average length of incomplete variants (generated in PART 2 of the algorithms) being smaller for $m_i > 1$, although it is true also that the amount of computation related to an incomplete variant grows with m_i since tuples from all m_i tables are under consideration when a test is performed. It was observed that the solution time increases with increase of the amount of incomplete variants generated. The amount of incomplete variants depends on properties of the matrix $\|a_{ij}\|_{m,n}$ when $m, n, b_i, i = 1, \dots, m$, are fixed. When

$(c_1/a_{i1}) \geq (c_2/a_{i2}) \geq \dots \geq (c_n/a_{in}), i = \overline{1, m}$, the algorithms are most efficient. The observed amount of tuples in a separate

table for such problems did not exceed $2\bar{z}$. The relatively small amount of tuples in this class of problems does not lead to a large amount of incomplete variants (which is the case when matrix $\|a_{ij}\|_{m,n}$ has a large amount of 0's).

It was also observed that algorithm 2 is far more efficient than algorithm 1 for problems where the amount of enumeration is large (sparse matrices $\|a_{ij}\|_{m,n}$ are often responsible for this).

To find the upper bound \bar{z} T algorithm 1 and algorithm 2 employ knapsack relaxation by using a surrogate constraint. Although the surrogate constraint producing the minimal upper bound may be found [17], it requires to solve a large linear programming problem. H. Pirkul had developed a relatively simple iterative algorithm for finding an approximation to the optimal surrogate constraint. This relaxation is implemented in [1] and [2].

Lagrangian relaxation approach was successfully used by Held and Karp [13], [14] and other authors [3], [4], [12] for special classes of problems. In this approach the problem (2) may be viewed as problem (3) complicated by m Knapsack constraints from (2). Moving the m constraints into Lagrangian function produces a Lagrangian relaxation of problem (2) in the form of problem (3).

IMPLEMENTATION OF LAGRANGIAN RELAXATION BY S.LEBEDEV ([16], [17])

Lebedev had combined knapsack and Lagrangian relaxations in one computational routine producing a bound better or not worse than delivered by any of them separately. For a problem of linear integer programming

$$L(x) = \sum_{j=1}^n c_j x_j \rightarrow \max \quad (6)$$

$$\sum_{j=1}^n a_{ij} x_j \leq b_i, i = 1, \dots, m; \quad (7)$$

$$0 \leq x_j \leq d_j, j=1, \dots, n; \quad (8)$$

$$x_j - \text{integers}, j=1, \dots, n \quad (9)$$

a set $T(\mathbf{I}) = \{x = (x_1, x_2, \dots, x_n) \mid \sum_{j=1}^n a_{ij} x_j \leq b_i, 0 \leq x_j \leq d_j, x_j - \text{integers}, j=1, \dots, n\}$,

where $a_j = \sum_{i=1}^m a_{ij} I_i, b = \sum_{i=1}^m b_i I_i, I_i \geq 0, i=1, \dots, m$ with $\sum_{i=1}^m I_i = 1$, is defined. Let S be the set of

feasible solutions of (6) - (9). Including knapsack constraints in (7) with nonnegative multipliers u_1, u_2, \dots, u_m , components of vector u , in Lagrangian function we receive

$$\max_{x \in S} L(x) = \max_{x \in T(\mathbf{I})} \min_u [L(x) + \sum_{i=1}^m r_i(x) u_i],$$

where $r_i(x) = b_i - \sum_{j=1}^n a_{ij} x_j, i = 1, \dots, m$. In what follows u and \mathbf{I} remain vectors of nonnegative components.

Exchanging the order of max and min operations results in

$$\max_{x \in S} L(x) \leq \min_u \max_{x \in T(\mathbf{I})} L(x, u)$$

where $L(x,u) = L(x) + \sum_{i=1}^m r_i(x)u_i$ (the difference between the right and left sides of the inequality is the so-called duality gap). Since $L(x) \leq L(x,u)$ for any $x \in S$ the problem

$$\max_{x \in T(I)} L(x,u)$$

produces an upper bound which may be used instead of \bar{z} in algorithm 1 and algorithm 2. It is desirable to decrease this upper bound by minimizing

$$j(u, I) = \max_{x \in T(I)} L(x,u).$$

The function $j(u, I)$ is a continuous convex piecewise-linear function for a fixed $I = I^0$, therefore it has a derivative in any direction. Vector u^* , such that $j(u^*, I^0) = \min_u j(u, I^0)$, is the vector of Generalized Lagrangian

Multipliers (GLM). Let u be an arbitrary point and $X(u)$ be the set of optimal solutions of the Knapsack problem $j(u, I^0) = \max_{x \in T(I^0)} L(x,u)$ and the derivative of $j(u, I^0)$ in direction w^0 , ($|w^0| = 1$), is being computed at point

u^0 . Let $u^1 = u^0 + hw^0$, where h is a small positive number, and $w_i^0 \geq 0$, if $u_i^0 = 0$. For small enough h

$X(u^1) \subseteq X(u^0)$ since for $x^t \notin X(u^0)$ the $L(x^t, u^0) \equiv L(x^t) + \sum_{i=1}^m r_i(x^t)u_i^0 < j(u^0, I^0)$ and, as the result of

$j(u, I^0)$ and $L(x,u)$ both being continuous with respect to u ($L(x,u) \equiv L(x) + \sum_{i=1}^m r_i(x)u_i$), such an h

may be chosen that $L(x^t, u^1) < j(u^1, I^0)$.

Then $j(u^1, I^0) = \max_{x \in T(I^0)} [L(x, u^0) + h \sum_{i=1}^m r_i(x)w_i^0] = L(x^t, u^0) + h \sum_{i=1}^m r_i(x^t)w_i^0$

for any $x^t \in X(u^1)$ and $j(u^0, I^0) - j(u^1, I^0) = \sum_{i=1}^m r_i(x^t)w_i^0$ for $x^t \in X(u^0 + hw^0)$ or (what is the same)

$\partial j(u^0, I^0) / \partial w^0 = \max_{t | x^t \in X(u^0)} \sum_{i=1}^m r_i(x^t)w_i^0$. Notice that $X(u^1)$ includes those and only those solutions from

$X(u^0)$ for which the maximum value is $\sum_{i=1}^m r_i(x^t)w_i^0$. Although these considerations allow to produce a method

for decreasing $j(u, I^0)$ by selecting a w_0 -direction with a negative derivative and changing one coordinate of u at a time, it doesn't produce the GLM-components of the vector minimizing $j(u, I^0)$. Generalized Lagrangian Multipliers $u_i^1, i=1,2,\dots,m$, may be found as components of an optimal solution of a linear programming problem. Its constraints are received by utilizing solutions of $X(u^0)$. The number of such constraints may be very large but by applying Dantzig-Wolfe decomposition they are generated only when needed. The Knapsack problem

$\max_{x \in T(I)} L(x, u^1) = \max \left(\sum_{j=1}^n (c_j - \sum_{i=1}^m a_{ij}u_i^1)x_j + \sum_{i=1}^m b_i u_i^1 \right) | \sum_{j=1}^n a_{ij}x_j \leq b, 0 \leq x_j \leq d_j, x_j - \text{integers}, j=1,\dots,n$

may be solved by a modified version of algorithm 1 or algorithm 2 which allow for real positive coefficients for the objective function and nonnegative integer components of vector x . It is the inclusion of a surrogate Knapsack

constraint $\sum_{j=1}^n a_{ij}x_j \leq b$ for improving the bound when solving a general integer linear problem constitutes the

main difference between approaches [3], [4], [12], [13], [14] and Lebedev's approach. Starting with some \mathbf{I}^0 the GLM vector \mathbf{u}^1 is computed and used for finding vector \mathbf{I}^1 by solving the minimization problem $\min_{\mathbf{I}} \mathbf{j}(\mathbf{u}^1, \mathbf{I})$.

This is followed by solving the problem $\min_{\mathbf{u}} \mathbf{j}(\mathbf{u}, \mathbf{I}^1)$ and obtaining next GLM vector \mathbf{u}^2 and so on. To solve the problem $\min_{\mathbf{I}} \mathbf{j}(\mathbf{u}^1, \mathbf{I})$ another special problem of linear programming may be solved. This process is finite and delivers $\mathbf{j}(\hat{\mathbf{u}}, \hat{\mathbf{v}})$ - a bound better or not worse than the bounds produced by knapsack and Lagrangian relaxations $\mathbf{j}(0, \mathbf{I}^*)$ or $\mathbf{j}(\mathbf{u}^*, 0)$ correspondingly. The process does not guarantee convergence to the optimal value for $\mathbf{j}(\mathbf{u}, \mathbf{I})$ (thus not allowing to compute the size of the duality gap). The use of Generalized Lagrangian Multipliers is especially suitable for linear programming problems of distributive type with integer coefficients.

PARALLELIZATION OF ORDERED ENUMERATION ALGORITHMS [11], [2], [1]

Ordered enumeration algorithms for a multidimensional 0/1 Knapsack problem have a high degree of parallelism and exploiting it can very substantially reduce the solution time [11]. Computations of triples in algorithm 1 and algorithm 2 are performed for each of m_1 tables sequentially. Numerical experiments [7] show that increasing m_1 reduces the processing time considerably. It is due to the fact that many incomplete variants constructed in PART 2 either get shorter or disappear when $m_1 > 1$. When constructing a solution in PART 2 the test for each x_k , $k \leq n$, is performed also sequentially. Both computations for each table $i, i=1, 2, \dots, m_1$, and assignment test of x_k for each $i, i=1, 2, \dots, m_1$, can be done in parallel. Even larger improvements of algorithm 1 can be expected by generated solutions for each z , $\hat{z} \leq z \leq \bar{z}$, in parallel. This conclusion follows from numerical experiments with problems where the amount of enumeration was large. Algorithm 2 was far more efficient than algorithm 1 for these problems. It is explained by the fact that algorithm 2 does not repeat construction of the same incomplete variants for $z < \bar{z}$. Generating solutions for each z , $\hat{z} \leq z \leq \bar{z}$, in parallel produces an even stronger impact since no delays for finding solutions for $z < \bar{z}$ are needed. Further parallelization of algorithm 1 can be achieved by storing lists $L(z)$, $\hat{z} \leq z \leq \bar{z}$, of variants to be continued with another assignment of the last variable. If both assignments (0 and 1) are possible for $x_k, k < n$, variant $(x_1, x_{n-1}, \dots, x_k=1)$, being constructed for $z=z_0$, is stored in $L(z_0)$. At any time available processors remove variants from $L(z_0)$ and continue their construction. Each such construction makes use of m processors. There is no need in keeping list L if lists $L(z)$, $\hat{z} \leq z \leq \bar{z}$, are maintained.

The m_1 tables constructed in PART 1 contain many tuples which are not needed for producing the optimal solution. Their presence makes the number of constructed incomplete variants large. It can be reduced by constructing additional m_1 tables for the same variables taken in reverse order. These additional m_1 tables are used to purge the

original ones. Let \hat{z} be a value of $\sum_{j=1}^n c_j x_j$ on a feasible solution of (2) and $[k^{(i_0)}, w_1^{(i_0)}]$ a tuple in column

z_0 of table i_0 from the first set of tables. If no tuple $[j^{(i_0)}, w^{(i_0)}]$ from the second set of tables for $z, z > \hat{z} - z_0$ satisfies conditions

$$j^{(i_0)} > k^{(i_0)}, w^{(i_0)} + w_1^{(i_0)} \leq b_{i_0}$$

then $[k^{(i_0)}, w^{(i_0)}]$ and any tuple $[k^{(i)}, w_1^{(i)}]$ from column z_0 in the first set of tables, such that $k^i = k^{i_0}, i=1, 2, \dots, m_1$

can be discarded also. The 2 sets of tables may be constructed at the same time and a number of tuples can be tested in parallel. Two parallel ordered enumeration algorithms and their implementation on a parallel system may be found in [1], [2].

REFERENCES

- [1] Alfred G. Burns, A Parallel Algorithm for the Multidimensional Knapsack Problem, Master's Thesis, Computer Science Department of ESU, 1993.
- [2] Alfred G. Burns and Felix Friedman, Parallel Ordered Enumeration Algorithms for the Multidimensional Knapsack Problem on Transputer Networks, Proceedings of the Seventh Conference of the North American Transputer Group, IOS Press, 1995.
- [3] H. Everett, Generalized Lagrange Multiplier Method for Solving Problems of Optimum Allocation of Resources, Operations Res., Vol. 11 (1963), pp.399-417.
- [4] M. L. Fisher, The Lagrangian Relaxation Method for Solving Integer Programming Problems, Management Science Vol.27, No.1, (Jan 1981), pp.1-18.
- [5] F. Friedman, Ordered Enumeration Algorithm for Linear Integer Programming, Joint National ORSA/TIMS Meeting, Orlando, Florida, 1983.
- [6] F. Friedman, Ordered Enumeration Method for Integer Programming, Journal of Economics and Mathematical Methods, Vol. 10, Issue 5, U.S.S.R Academy of Sciences, Moscow, 1974, pp. 964-967 (in Russian).
- [7] F. Friedman, Two Modifications of Ordered Enumeration Method and FORTRAN Programs, Algorithms and Programs, Issue 60, U.S.S.R. Academy of Sciences, Moscow, 1974 pp. 3-70 (in Russian).
- [8] F. Friedman and S. S. Lebedev, Modified Ordered Enumeration Method, Summary of Reports on Third All-Union Symposium Programming Systems for Solving Planning Optimization Problems, U.S.S.R. Academy of Sciences, Moscow 1974, pp. 52-54 (in Russian).
- [9] F. Friedman, Knapsack Problem with Special Additional Constraints, Proceedings of 6th Winter School for Mathematical Programming and Related Problems (Drogobich), U.S.S.R. Academy of Sciences, Moscow, 1975, pp. 244-258 (in Russian).
- [10] F. Friedman, Numerical Experiment for Ordered Enumeration Method, Summary of Reports on 4th All-Union Symposium Programming Systems for Solving Planning Optimization Problems, U.S.S.R. Academy of Sciences, Moscow, 1976, pp.32-35 (in Russian).
- [11] F. Friedman, Ordered Enumeration Algorithms: Extended Applications and Parallelization, Computer Science Department of East Stroudsburg University, Technical Report, 1988.
- [12] A. M. Geoffrion, Lagrangian Relaxation and Its Uses in Integer Programming, Mathematical Programming Study, Vol. 2, 1974, pp.40-47.
- [13] M. Held and R. M. Karp, The Travelling Salesman Problem and Minimum Spanning Trees, Operations Research, Vol. 18, 1970, pp.1138-1162.
- [14] M. Held and R. M. Karp, The Travelling Salesman Problem and Minimum Spanning Trees: Part II, Mathematical Programming, Vol. 1, 1971, pp.6-25.
- [15] S.S. Lebedev, Combinatorial Methods of Discrete Programming, Proceedings of the 1968 Alma-Ata Summer School on Mathematical Programming, U.S.S.R. Academy of Sciences, Alma-Ata, 1969, pp.38-88 (in Russian).
- [16] S. S. Lebedev, Use of Generalized Optimal Multipliers for Solving Linear Integer Programming Problems by Ordered Enumeration Method, Journal of Mathematical Methods in Economics, Nauka, Moscow, 1974, pp.70-78 (in Russian).
- [17] S. S. Lebedev, Generalized Lagrangian Multipliers in Integer Programming, Proceedings of 6th Winter School for Mathematical Programming and Related Problems (Drogobich), U.S.S.R. Academy of Sciences, Moscow, 1975, pp.114-157 (in Russian).
- [18] T. Lee, E. Shragowitz, and S. Sahni, A Hypercube Algorithm for the 0/1 Knapsack Problem, Proceedings of the International Conference on Parallel Processing, 1986, pp.699-706.
- [19] Prabhakant, Sinha and A. A. Zoltners, The Multiple-choice Knapsack Problem, Operations Research, Vol.27, No.3, May-June 1979, pp.503-513.