An exact algorithm for the subset-sum problem

Hiroshi IIDA
e-mail : *x-iida@jaist.ac.jp*

Milan Vlach
e-mail : *vlach@jaist.ac.jp*

School of Information Science,
Japan Advanced Institute of Science and Technology, Hokuriku†

Abstract

In this paper we propose a new algorithm for solving the subset-sum problem. First we propose a new algorithm (XS-algorithm) for the partition problem. Then we describe a transformation of the subset-sum problem to the partition problem, and show how to solve the resulting problem by a slightly modified version of XS-algorithm. Finally, we present results of extensive computational experiments for several types of data instances. The paper is based on the master thesis of the first author.

1 Introduction

The main purpose of this paper is to present a new algorithm for solving the Subset-Sum Problem. The Subset-Sum Problem (SSP) can formally be stated as follows: Given $n+1$ positive integers $a_1, a_2, \ldots, a_n$ and c,

$$\text{maximize} \quad \sum_{j=1}^{n} a_j x_j \tag{1}$$

$$\text{subject to} \quad \sum_{j=1}^{n} a_j x_j \leq c \tag{2}$$

$$x_j \in \{0,1\}, \quad 1 \leq j \leq n. \tag{3}$$

The SSP can be considered as a special case of the classical 0–1 Knapsack Problem, which can be formulated as follows: Given $2n$ positive integers p_j, w_j $(1 \leq j \leq n)$ and a positive integer c,

$$\text{maximize} \quad \sum_{j=1}^{n} p_j x_j$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \leq c$$

$$x_j \in \{0,1\}, \quad 1 \leq j \leq n.$$

†1–1 Asahidai Tatsunokuchi Ishikawa 923–12, JAPAN.

Following the terminology of Martello & Toth's book on knapsack problems [3], we shall sometimes call the elements of the set $\{1, 2, \ldots, n\}$ items. The numbers p_j and w_j associated with each item j will be called *profit* and *weight* of item j, respectively. The number c will be called capacity.

Obviously, if p_j equals w_j for every item $j \in N$, then the 0–1 Knapsack Problem reduces to the SSP. Thus the SSP can informally be stated as the problem of selecting a subset of items such that the sum of the weights of the selected items is maximized without exceeding a given capacity c.

Our approach to solving the SSP is based on an algorithm for solving the following Partition Problem (PP): Given n positive integers $c_1, c_2, \ldots, c_n$,

$$\text{minimize} \quad \left| \sum_{j \in J} c_j - \sum_{j \notin J} c_j \right| \tag{4}$$

$$\text{subject to} \quad J \subset \{1, 2, \ldots, n\} \tag{5}$$

This new algorithm is described in detail in section 2. In section 3, we study relations between the SSP and PP, and we show how to solve the SSP with the help of a modification of the proposed algorithm for PP. Section 4 provides results of extensive computational experiments for several types of standard data instances. The last section is devoted to conclusions.

2 An algorithm for the partition problem

In this section we describe a new algorithm for solving the Partition Problem. Before describing the details of the algorithm, which we call XS-algorithm or XS simply, we introduce some notations and definitions.

Let $N := \{1, 2, \ldots, n\}$ and let J be a subset of N. The complement of J in N will be denoted by $\bar{J}$, and the number of elements of J will be called *length* of J. In addition, we denote the length of J as $|J|$. With each $J \subset N$, we associate a 0–1 n-vector $x^J = (x_1^J, x_2^J, \ldots, x_n^J)$ by defining

$$x_i^J := \begin{cases} 1 & \text{if } i \in J \\ 0 & \text{if } i \notin J \end{cases}$$

The vector $(x_n^J, x_{n-1}^J, \ldots, x_1^J)$ will be called *bit pattern* of J. On the other hand, with each 0–1 n-vector $x = (x_1, x_2, \ldots, x_n)$ we associate a subset J_x of N by the condition: $j \in J_x$ if and only if $x_j = 1$. For each $J \subset N$ and a family $\{c_j : j \in N\}$, the sum $\sum_{j \in J} c_j$ will be denoted by $c(J)$, and called *weight* of J. The number $\left| c(J) - c(\bar{J}) \right|$ will be called *cost* of J.

The PP is obviously an $\mathcal{NP}$-hard problem, because the problem of deciding whether there exists $J \subset N$ such that $c(J) = c(\bar{J})$ is an $\mathcal{NP}$-complete problem — see PARTITION in Karp [1] or Garey & Johnson [2]. However, the PP has special structural properties that make it possible to solve relatively large problems in reasonable time. Our approach to solving the PP is based on the classical depth-first tree search. The proposed algorithm consists of two main parts: preprocessing and searching. The purpose of preprocessing is to construct trees for searching, and an incumbent initial solution and value.

2.1 Preprocessing

From now on, we shall always assume that the items are ordered according to nondecreasing values of their weights, i.e. so that $c_1 \leq c_2 \leq \cdots \leq c_n$. The preprocessing performed by the XS-algorithm is based on the following three observations.

Observation 1 Let k be the item such that

$$\sum_{j=n-k+2}^{n} c_j \leq \frac{c(N)}{2} < \sum_{j=n-k+1}^{n} c_j$$

Since $c_n + c_{n-1} + \cdots + c_{n-k+2}$ is the largest sum among all sums corresponding to the subsets the lengths of which are less than k, we can discard from consideration all subsets J with $|J| < k$ except $\{n, n-1, \ldots, n-k+2\}$. We take this set as an initial incumbent solution, and its cost as an initial incumbent value of the objective function (4).

Observation 2 We can discard all subsets J such that $|J| > \lfloor n/2 \rfloor$, because the cost of each set J the length of which is greater than $\lfloor n/2 \rfloor$ is equal to the cost of $\bar{J}$ and the length of $\bar{J}$ is less than or equal to $\lfloor n/2 \rfloor$.

Observation 3 If n is even, we can fix an arbitrary item first, and then we can discard all subsets J of the length $n/2$ which contain the fixed item. For example, if $n = 4$ and if we fix item 4, then we can discard the sets $\{1,4\}, \{2,4\}, \{3,4\}$, because their costs are equal to those of complements $\{2,3\}, \{1,3\}, \{1,2\}$, respectively.

In accordance with Observations 1 and 2, we can restrict further searching to the subsets of N the length of which varies from k to $\lfloor n/2 \rfloor$. For every such length, the XS-algorithm creates a binary tree for enumerating items in the order of nonincreasing weights, i.e. in the order $n, n-1, \cdots$ see Figure 1. If n is even, then the tree corresponding to the set-length of $n/2$ can be simplified by taking into consideration Observation 3. To be specific, we always fix item n, and discard all subsets of the length $n/2$ containing item n. In other words, at the root of the corresponding tree is item $n-1$ instead of item n.

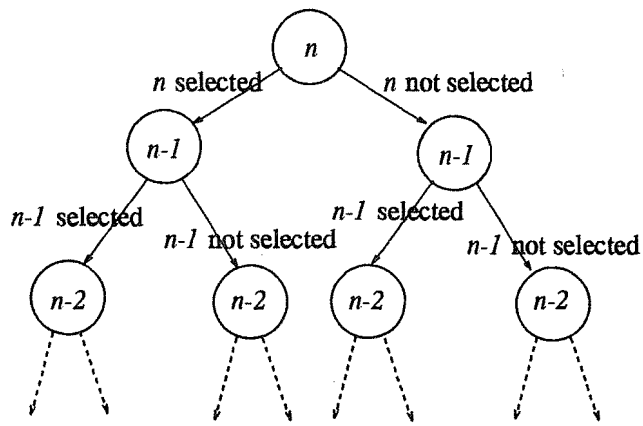

Figure 1: search tree

2.2 Searching

Searching for an optimal solution is based on the following three procedures, which we call TEST, LAST and SKIP. From the preprocessing we know that we have to search in $\lfloor n/2 \rfloor - k+1$ trees, and we also have already the incumbent solution $J = \{n, n-1, \ldots, n-k+2\}$ and its cost. Let us briefly describe the procedures mentioned above. In subsection 3.3, we present a sample implementation for the routine of tree search taking these three procedures into consideration.

TEST Suppose we are at a node of one of the search trees. Let l be the length of solutions prescribed for the tree under consideration, and let the item of the node be j. In order to decide whether to move along "j-selected" edge or "j-not selected" edge, we perform the following test, provided the number of items already selected is less than $l-1$. Let J be the set of already selected items. We add item j to J together with items $1, 2, \ldots, i$ where i is such that the length of this new set $J_1 := J \cup \{j, 1, 2, \ldots, i\}$ is equal to the prescribed length l. Then we compute the difference $c(J_1) - c(\bar{J}_1)$ and test whether it is greater than or equal to the incumbent value. If so, then obviously the cost of all subsets represented by the node resulting from selecting item j is also greater than or equal to the incumbent value. Therefore we cut that part and move along the "j-not selected" edge. If the difference is less than the incumbent value, then we move along the "j-selected" edge.

LAST Suppose we are at a node such that the length of the corresponding set of selected items J is one unit shorter than the prescribed length. Let $j \in J$ be the item of the minimum weight among items in J. If $c(J) + c_1 \geq c(N)/2$, then we add item 1 to J, and go directly to the corresponding node, and perform BACKTRACK. We shall discuss the procedure BACKTRACK later. If $c(J) + c_{j-1} < c(N)/2$, then we move to the node corresponding to the set $J \cup \{j-1\}$ along the "$j-1$ selected" edge, and perform SKIP. If $c(J) + c_1 < c(N)/2 \leq c(J) + c_{j-1}$, then we perform a binary search to find item p such that

$$1 \leq p < j-1 \quad \text{and} \quad c(J) + c_p < \frac{c(N)}{2} \leq c(J) + c_{p+1}.$$

Then we add item p to J and go directly to the corresponding node, and perform SKIP.

SKIP Suppose we are at a node at which the length of sets represented by the node is equal to the prescribed length. Let J be the corresponding set of already selected items. If $c(J) \geq c(N)/2$, BACKTRACK is performed. If $c(J) < c(N)/2$, then we need not continue searching from this node. Therefore we skip to some suitable node or, if impossible, we terminate the search in the tree. We shall describe the skipping procedure by means of an example. Consider the situation depicted in Figure 2. First note that the partial bit patterns

$$\begin{aligned} \alpha &: \texttt{0101111xx} \\ \beta &: \texttt{01011101x} \\ \gamma &: \texttt{00xxxxxx} \end{aligned}$$

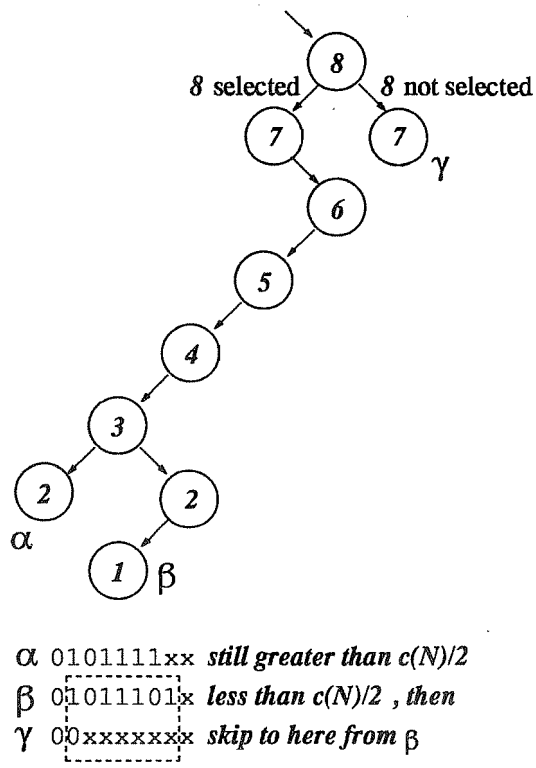

Figure 2: SKIP

represent the collections of subsets determined by the nodes labelled also by α, β and γ, respectively. Let J_α, J_β and J_γ denote the corresponding sets of selected items, i.e. $J_\alpha = \{8, 6, 5, 4, 3\}$, $J_\beta = \{8, 6, 5, 4, 2\}$, $J_\gamma = \emptyset$. Let us assume that

$$c(J_\beta) < \frac{c(N)}{2} \leq c(J_\alpha).$$

If we skip from node β to node γ, then we can possibly reach from node γ better solutions. Since each node has its own unique partial bit pattern, we can realize skipping very efficiently. In the example, we attempt to find first '1' after skipping over two (or more if necessary) '0' to the left from the position indicating the last-selected item. Then we replace the found '1' by '0' and leave free all other bits to the right. For instance, in Figure 2, we assume that SKIP is performed at the node labelled β. In this case we obtain the node labelled γ, and continue the search. In this case we have skipped over exactly two '0' (c_3, c_7). If we could not skip over at least two '0' or could not find '1' after skipping over two (or more) '0', processing of the tree terminates.

Finally we mention the procedure BACKTRACK. Roughly speaking, BACKTRACK is similar to SKIP. The only essential difference is that in BACKTRACK we attempt to find the first '1' after skipping over one (or more if necessary) '0' instead of at least two '0' from the position indicating the last-selected item.

Remark In order to enhance the efficiency of searching, we go directly to the node corresponding to the set $J_1 := J \cup \{1, 2, \ldots, j-1\}$ whenever $l - |J| = j - 1$, where

l is the prescribed length and j is the item of the minimum weight in J. Then SKIP is performed whenever $c(J_1) < c(N)/2$. If $c(J_1) \geq c(N)/2$, then BACKTRACK is performed.

3 Subset-sum problem

To begin with, we show that the PP is equivalent to a special structured SSP. Therefore the XS-algorithm described in the previous section is directly applicable to certain instances of the general SSP.

Proposition 1 A subset J of N is an optimal solution of the Partition Problem (4)–(5) if and only if the corresponding x^J is optimal for the problem

$$\text{maximize} \quad \sum_{j=1}^{n} c_j x_j \tag{6}$$

$$\text{subject to} \quad \sum_{j=1}^{n} c_j x_j \leq \left\lfloor \frac{1}{2} \sum_{j=1}^{n} c_j \right\rfloor \tag{7}$$

$$x_j \in \{0,1\},\ j \in N \tag{8}$$

Proof Suppose that J is an optimal solution of (4)–(5). Without loss of generality we may assume that $c(J) \leq c(\bar{J})$. It follows that $c(J) \leq \frac{1}{2}c(N)$, because $c(J)+c(\bar{J}) = c(N)$. Since $c(J)$ is an integer, we have also

$$c(J) \leq \left\lfloor \frac{1}{2}c(N) \right\rfloor .$$

Since $c(J) = \sum_{j=1}^{n} c_j x_j^J$, we see that x^J is a feasible solution of (6)–(8). Now we verify that x^J is an optimal solution of (6)–(8). Suppose that x^J is not optimal. Then there exists $K \subset N$ such that

$$\sum_{j=1}^{n} c_j x_j^J < \sum_{j=1}^{n} c_j^K \leq \left\lfloor \frac{1}{2}c(N) \right\rfloor$$

This is equivalent to

$$c(J) < c(K) \leq \left\lfloor \frac{1}{2}c(N) \right\rfloor$$

Taking into consideration that $c(\bar{K}) \geq c(K)$, we obtain

$$\begin{aligned} c(\bar{J}) - c(J) &> c(\bar{J}) - c(K) \\ &= c(\bar{J}) + c(J) - c(J) - c(K) + c(\bar{K}) - c(\bar{K}) \\ &= c(N) - c(J) - c(N) + c(\bar{K}) = c(\bar{K}) - c(J) \\ &> c(\bar{K}) - c(K) \geq 0 \end{aligned}$$

and consequently

$$\left| c(\bar{J}) - c(J) \right| > \left| c(\bar{K}) - c(K) \right| ,$$

which contradicts the optimality of J for (4)–(5). Thus x^J must be optimal.

Now we prove the converse implication. Let $x = (x_1, x_2, \ldots, x_n)$ be an optimal solution of (6)–(8). We now show that J_x is optimal for (4)–(5). Suppose this is not true. Then there exists $J \subset N$ such that $c(J) \leq \left\lfloor \frac{1}{2} c(N) \right\rfloor$ and

$$c(\bar{J}_x) - c(J_x) > c(\bar{J}) - c(J) \geq 0.$$

However

$$\begin{aligned} c(J) - c(J_x) &= c(N) - c(\bar{J}) - (c(N) - c(\bar{J}_x)) \\ &= c(\bar{J}_x) - c(\bar{J}) > c(J_x) - c(J) \end{aligned}$$

It follows that

$$c(J_x) < c(J) \leq \left\lfloor \frac{1}{2} c(N) \right\rfloor$$

However this is equivalent to

$$\sum_{j=1}^{n} c_j x_j < \sum_{j=1}^{n} c_j x_j^J \leq \left\lfloor \frac{1}{2} c(N) \right\rfloor,$$

which contradicts the optimality of x. ∎

Proposition 1 shows that every algorithm for solving the PP can be used for solving special cases of the SSP given by (6)–(8). However, it is not clear how to apply the XS-algorithm to the instances of SSP with $c \neq \left\lfloor \frac{1}{2} \sum_{j=1}^{n} a_j \right\rfloor$. The following simple example demonstrates that the XS-algorithm may fail in delivering an optimal solution of such instances, if applied to the PP with $c_j := a_j$ for all $j \in N$.

Example 1 Consider the instance of the SSP given by $n = 3, a_1 = 1, a_2 = 2, a_3 = 3, c = 4$. Its unique solution is $x_1 = 1, x_2 = 0, x_3 = 1$. It is obvious that the XS-algorithm cannot deliver this solution if applied to the instance of the PP given by $n = 3, c_1 := a_1, c_2 := a_2, c_3 := a_3$, because the only optimal solutions of this instance are $\{1, 2\}$ and $\{3\}$.

In order to overcome this difficulty we propose such conversion of the general SSP to the PP that a modification of the XS-algorithm can solve the original problem.

3.1 Conversion from the SSP to PP

The main problem of conversion is caused by the fact that, in general, the right hand side c of (2) may differ from $\left\lfloor \frac{1}{2} \sum_{j=1}^{n} a_j \right\rfloor$ as required by (7). To cope with this we suggest to transform the general SSP with n items given by (1)–(3) to the PP with $n+2$ numbers $c_1, c_2, \ldots, c_{n+2}$ defined by

$$\begin{cases} c_j = a_j & \text{for all } j \in N \\ c_{n+1} = 2c & \\ c_{n+2} = \sum_{j=1}^{n} a_j & \end{cases} \tag{9}$$

Note that if $\sum_{j\in J} a_j = c$ for some $J \subset N$, then $c(J \cup \{n+2\})$ is exactly the half of $c(J \cup \{n+1, n+2\})$. Similarly to the PP, we denote $\sum_{j\in J} a_j$ by $a(J)$ and call it weight of J.

Remark This conversion differs slightly from the classical conversion of Karp [1] given by

$$\begin{cases} c_j &= a_j & \text{for all } j \in N \\ c_{n+1} &= c+1 & \\ c_{n+2} &= \sum_{j=1}^{n} a_j + 1 - c & \end{cases}$$

We replace "1" in the Karp conversion by "c" in order to prevent the subsets of $N \cup \{n+1, n+2\}$ containing either both $n+1$ and $n+2$ or none of them from being optimal for the resulting PP. Our conversion differs also from the Vavasis conversion [5] defined by

$$\begin{cases} c_j &= a_j & \text{for all } j \in N \\ c_{n+1} &= \sum_{j=1}^{n} a_j - 2c & \end{cases}$$

in which c_{n+1} may become negative.

Proposition 2 A vector $x = (x_1, x_2, \ldots, x_n)$ is an optimal solution of the problem

$$\text{minimize} \quad \left| c - \sum_{j=1}^{n} a_j x_j \right| \tag{10}$$

$$\text{subject to} \quad x_j \in \{0,1\}, \quad \text{for all } j \in N \tag{11}$$

if and only if $J_x \cup \{n+2\}$ is an optimal solution of the PP with data (9).

Proof Let J be an arbitrary subset of N. Let us calculate the values of the objective function of the PP under consideration for $J \cup \{n+1, n+2\}$ and $J \cup \{n+2\}$. For the former, we have

$$\begin{aligned} \left| a(J) + 2c + a(N) - a(\bar{J}) \right| &= \left| a(J) + 2c + a(J) + a(\bar{J}) - a(\bar{J}) \right| \\ &= 2\,|\,a(J) + c\,|\,, \end{aligned}$$

and for the latter, we obtain

$$\begin{aligned} \left| a(J) + a(N) - a(\bar{J}) - 2c \right| &= \left| a(J) + a(J) + a(\bar{J}) - a(\bar{J}) - 2c \right| \\ &= 2\,|\,a(J) - c\,|\,. \end{aligned}$$

Consider the partition of the set of all subsets of $N \cup \{n+1, n+2\}$ into the four sets S_1, S_2, S_3 and S_4 defined as follows:

S_1 is the set of all subsets containing $n+2$ and not containing $n+1$,

S_2 is the set of all subsets containing $n+1$ and not containing $n+2$,

S_3 is the set of all subsets containing both $n+1$ and $n+2$,

S_4 is the set of all subsets containing neither $n+1$ nor $n+2$.

Since $|a(J)+c| \geq |a(J)-c|$ for all $J \subset N$ and since S_2 is exactly the set of all complements of the sets from S_1 and S_4 is exactly the set of all complements of the sets from S_3, we conclude that if x is an optimal solution of (10)–(11), then $J_x \cup \{n+2\}$ solves the corresponding PP.

Now suppose that x is not optimal for (10)–(11). Then there is x' such that

$$\left| c - \sum_{j=1}^{n} a_j x'_j \right| < \left| c - \sum_{j=1}^{n} a_j x_j \right|$$

Therefore $|c - a(J_{x'})| < |c - a(J_x)|$. The values of the objective function for the corresponding PP at $J_{x'} \cup \{n+2\}$ and $J_x \cup \{n+2\}$ are $2|a(J_{x'}) - c|$ and $2|a(J_x) - c|$, respectively. Therefore $J_x \cup \{n+2\}$ is not optimal. ∎

Remark Since S_2 is the set of all complements of the sets from S_1, it is obvious that a vector $x = (x_1, x_2, \ldots, x_n)$ is an optimal solution of (10)–(11) if and only if $J_x \cup \{n+1\}$ is an optimal solution of the PP with data (9).

The following example shows that an optimal solution of the PP defined by (9) is not necessarily an optimal solution of the original SSP. The reason is that an optimal solution x of (10)–(11) may be infeasible for the original SSP, because it may not satisfy the inequality $\sum_{j=1}^{n} a_j x_j \leq c$.

Example 2 Consider the instance of SSP given by $n = 3, a_1 = 3, a_2 = 4, a_3 = 8$ and $c = 10$. Using the transformation (9), we obtain the following instance of PP: $c_1 = 3, c_2 = 4, c_3 = 8, c_4 = 20, c_5 = 15$. The optimal solution containing item $n+2$ is $\{1, 3, 5\}$. However, $a_1 + a_3 = 11 > 10 = c$. Therefore $\{1, 3\}$ is infeasible for the original instance of SSP, and it cannot be optimal.

3.2 Modification of the xs-algorithm

Consider again an arbitrary instance of SSP given by $a_1, a_2, \ldots, a_n$ and c. Without any loss of generality, we assume that

- $a_1 \leq a_2 \leq \cdots \leq a_n$
- $a_1 + a_2 + \cdots + a_n > c$
- $a_j < c$ for all $j \in N$
- $a(N) \neq 2c$ and $a(N) \neq 2c+1$

Note that if the last assumption is not fulfilled, then $c = \lfloor a(N)/2 \rfloor$, and we can solve such problems by directly appling the unmodified XS-algorithm. In the terms of the PP defined by (9), the last assumption means that $c_{n+2} \neq c_{n+1}$ and $c_{n+2} \neq c_{n+1} + 1$.

It is obvious from Proposition 2 that if $J \subset N$ is such that $J \cup \{n+2\}$ solves the corresponding PP with data (9), then J solves the original SSP whenever $c(J) \leq c$. The previous example shows that the XS-algorithm may deliver an optimal solution with $c(J) > c$. To resolve this difficulty we modify the XS-algorithm as follows.

1. We discard all sets belonging to S_3 and S_4. Thus we deal only with subsets of $N' := N \cup \{n+1, n+2\}$ belonging either to S_1 or S_2.

2. To guarantee the feasibility, we discard all subsets containing item $n+2$ the weights of which are greater than $c(N')/2$ and all subsets containing item $n+1$ the weights of which are less than $c(N')/2$. The reason is that, for each $J \subset N$, $a(J) \leq c$ holds if and only if $c(J) + c_{n+2} \leq c(N')/2$, and $a(\bar{J}) \leq c$ holds if and only if $c(J) + c_{n+1} \geq c(N')/2$.

3. According to Observation 1 of subsection 2.1, the lower bound k of the subsets length is 2, because $\min\{c_{n+1}, c_{n+2}\} > a_j$ for each $j \in N$, $\max\{c_{n+1}, c_{n+2}\} \leq c(N')/2$, and $c_{n+1} + c_{n+2} > c(N')/2$. However, since we do not deal with subsets containing both items $n+1$ and $n+2$, we have to modify the definition of k. It is easy to see that the following modification will serve the purpose:

   (a) If $c_{n+2} > c_{n+1}$, we define k by

   $$c_{n+2} + \sum_{j=n-k+2}^{n} a_j > \frac{c(N')}{2} \geq c_{n+2} + \sum_{j=n-k+3}^{n} a_j, \quad 3 \leq k \leq n+1$$

   In this case, we may discard all subsets containing item $n+2$ the length of which are less than k, except $\{n+2, n, n-1, \ldots, n-k+3\}$. Since $c_{n+1} + a_n + a_{n-1} + \cdots + c_{n-k+3}$ is less than $c(N')/2$, we may discard also all subsets containing item $n+1$ the length of which are less than k. Consequently, if $c_{n+2} > c_{n+1}$, then we discard all subsets of N' the length of which are less than k, except $\{n+2, n, n-1, \ldots, n-k+3\}$. Then we take this set as initial incumbent solution and its cost as initial incumbent value.

   (b) If $c_{n+2} < c_{n+1}$, we define k by

   $$c_{n+1} + \sum_{j=n-k+2}^{n} a_j > \frac{c(N')}{2} \geq c_{n+1} + \sum_{j=n-k+3}^{n} a_j, \quad 2 \leq k \leq n \tag{12}$$

   In this case, we may discard all subsets containing item $n+1$ the length of which are less than k, except $\{n+1, n, \ldots, n-k+3\}$, and all subsets containing item $n+2$ the length of which are less than k except $\{n+2, n, n-1, \ldots, n-k+3\}$. Note that $\{n+1, n, \ldots, n-k+3\}$ cannot be discarded because its cost might be zero. Consequently, if $c_{n+2} < c_{n+1}$, then we discard all subsets of N' the length of which are less than k, except $\{n+1, n, \ldots, n-k+3\}$ and $\{n+2, n, n-1, \ldots, n-k+3\}$. If the cost of $\{n+1, n, \ldots, n-k+3\}$ is zero, then we found an optimal solution. If not, then we take $\{n+2, n, n-1, \ldots, n-k+3\}$ as incumbent solution.

4. As upper bound of the length of solutions we take $\lfloor (n+2)/2 \rfloor = \lfloor n/2 \rfloor + 1$.

5. If n is even and the prescribed length of subsets is $n/2$, we fix item $n+1$, provided $c_{n+1} > c_{n+2}$, or we fix item $n+2$, provided $c_{n+2} > c_{n+1}$, and we deal only with S_1 or S_2, respectively.

6. In order to obtain an optimal solution of the original SSP, we discard item $n+2$, provided that the modified XS-algorithm delivered an optimal solution containing $n+2$. If the XS-algorithm delivers an optimal solution containing $n+1$, we take its complement in N' and discard item $n+2$.

Let us summarize the proposed algorithm:

- Define $c_j := a_j$ for all $1 \leq j \leq n$ and sort them in a nondecreasing order, and rename (if necessary) so that $c_1 \leq c_2 \leq \cdots \leq c_n$.
- Construct two additional items $n+1$ and $n+2$ with weights $c_{n+1} := 2c$, $c_{n+2} := \sum_{j=1}^{n} a_j$.
- If $c_{n+2} = c_{n+1}$ or $c_{n+2} = c_{n+1} + 1$, then discard items $n+1$ and $n+2$, and solve the resulting PP by the unmodified XS-algorithm. If this results in an optimal solution J with $c(J) > c(\bar{J})$, then take $\bar{J}$ as an optimal solution of the original SSP.
- If $c_{n+2} \neq c_{n+1}$ and $c_{n+2} \neq c_{n+1}+1$, then we solve the PP with the help of the modified XS-algorithm. Then we take the solution containing item $n+2$ and discard $n+2$. The remaining set solves the original SSP.

3.3 Implementation

In this subsection we discuss an implementation of the basic routine, i.e. the routine for search in trees used for solving the PP.

Regarding the arguments of the routine: NI (Number of Items) varies from k to $\lfloor n/2 \rfloor$, BP (Base Position) is equal to n except for the case n even and NI$= n/2$. In the latter BP$=n-1$. If the routine is applied to an instance of SSP which requires the modified XS-algorithm, then c_{n+1} and c_{n+2} are managed separately, i.e. either item $n+1$ or item $n+2$ is already taken before invoking the routine.

```
INPUT : BP ; base position, the item of maximum number we could take
        NI ; # of items we should take

        if NI=1 then
                binary search in [1,BP], and find p; exit
        clear all L[]; // array for storing the selection of items
        j:=BP; L[j]:=1; rst:=NI-1;
eval:   if rst=j-1 then // no more room for selection
                take rest items as small number as possible and j:=1;
                if weight is less than c(N)/2 then
                        counter:=2 and goto back; // skip
                counter:=1, goto back;            // backtrack
        else if select test is N.G. then // testing item is 'j'
                ; // goto forw;
        else if rst=1 then
                binary search in [1,j-1], and find p;
```

```
                if p+2=j then
                        counter:=1 and goto back; // skip
                else if p+1=j then
                        counter:=2 and goto back; // skip
        else
                j--; L[j]:=1; rst--; goto eval;
forw:   L[j]:=0; j--; L[j]:=1; // forward move
        goto eval;
back:   L[j]:=0; j++; rst++;
        while j <= BP and counter > 0 do
                if L[j]=1 then
                        L[j]:=0; rst++;
                else
                        counter--;
                j++;
        // find '1' for forward move
        while j <= BP do
                if L[j]=1 then goto forw;
                j++;
        exit
```

In this pseudo-code, p indicates the item found by binary search in the range $[1, j-1]$. In particular, when the weight of subset $J \cup \{j-1\}$ is less than $c(N)/2$, then $p = j-1$, and when the weight of $J \cup \{1\}$ is greater than $c(N)/2$, then $p = 0$.

For the reader's convenience we also present a small-sized example. Consider an instance of the SSP with 6 items given by

a_6	a_5	a_4	a_3	a_2	a_1	c
14	10	8	6	4	2	23

In accordance with (9), the corresponding instance of the PP is given by

c_8	c_7	c_6	c_5	c_4	c_3	c_2	c_1
44	46	14	10	8	6	4	2

Since $c_8 \neq c_7$ and $c_8 \neq c_7 + 1$, we must apply the modified XS-algorithm. First, we determine the range for the length of solutions. Since $c_8 < c_7$, we determine k according to (12). Since

$$c_7 + c_6 + c_5 = 70 > 67 = \frac{c(N')}{2} \geq 60 = c_7 + c_6$$

we obtain $k = 3$. Since $\lfloor (n+2)/2 \rfloor = 4$, we have to search in two trees, taking the prescribed length of solutions 3 and 4. Before starting the main loop, we have to examine the subsets of the length 2, namely $\{7, 6\}$ and $\{8, 6\}$. For the sake of brevity, we shall represent subsets of J of N' by their bit patterns $(x_8^J x_7^J \cdots x_1^J)$, and the feasibility by the + sign in front of the pattern. In the case of subsets $\{7, 6\}$ and $\{8, 6\}$ we obtain

```
 01 100000 : 60, not equal to c(N')/2, discard it.
+10 100000 : 58, feasible, cost is 18(=|58-(134-58)|).
```

Now, we start the main loop. In the first tree, the prescribed length of solutions is three. First we consider solutions in S_2 (containing 7 and not containing 8), then solutions in S_1 (containing 8 and not containing 7). Thus, it is a good idea to duplicate the first tree. In the main loop, we first attempt to take item 6. According to the pseudo-code presented above, we perform TEST, and so on. As mentioned above, the items 7 and 8 are managed separately. In this way we obtain (the symbol x represents not yet decided positions):

```
 01 1xxxxx : 01 100001 -- c(J)=62, 62-72=-10<18, O.K. then, binary search
+01 101000 : 68, cost is 2, new incumbent (indeed optimal solution).
 01 100100 : 66, less than c(N')/2, skip.
 01 01xxxx : 01 010001 -- c(J)=58, 58-76=-18<2, O.K. then, binary search
 01 011000 : 64, less than c(N')/2, skip and end of 01xxxxxx
 10 1xxxxx : 10 100001 -- c(J)=60, 60-74=-14<2, O.K. then, binary search
 10 110000 : 68, over c(N')/2, then infeasible
+10 101000 : 66, skip and end of 10xxxxxx
```

The point is that the skip is not concerned with items 7 and 8. Next, we search the last tree in which we should take four items. Note that we take item 8 and discard item 7, because $c_8 < c_7$. In general, when n is even, the last tree is not duplicated.

```
 10 1xxxxx : 10 100011 -- c(J)=64, 64-70=-6<2,  O.K.
 10 11xxxx : 10 110001 -- c(J)=70, 70-64=6>=2,  N.G.
 10 101xxx : 10 101001 -- c(J)=68, 68-66=2>=2,  N.G.
 10 1001xx : 10 100101 -- c(J)=66, 66-68=-2<2,  O.K. then, binary search
 10 100110 : 68, over c(N')/2, then infeasible
+10 100101 : 66, skip
 10 01xxxx : 10 010011 -- c(J)=60, 60-74=-14<2, O.K.
 10 011xxx : 10 011001 -- c(J)=64, 64-70=-6<2,  O.K. then, binary search
 10 011100 : 68, over c(N')/2, then infeasible
+10 011010 : 66, skip and processing terminates.
```

Finally, we invert the bit pattern $(01101000)_2$ indicating an optimal solution and discard the bits representing items 7 and 8. We obtain $(010111)_2$ as an optimal solution of original SSP. Its weight is $c(\{1,2,3,5\}) = 2+4+6+10 = 22$.

4 Computational experiments

In order to test the efficiency of the proposed algorithm we compared it with the MTSL algorithm by Martello & Toth [3] and the NRSUB algorithm by Pisinger [4]. We have implemented all algorithms in C and examined their behaviour on several groups of instances used for experimenting both by Martello & Toth and Pisinger.

Remark When implementing MTSL and NRSUB, we modified them slightly. We believe that these modifications may improve their performance. First, in the MTS procedure of Martello & Toth, we change the condition in the beginning of block 3 [perform a forward move] from "$w_j \leq \hat{c}$ and $j < NA$ and $\hat{c} > \bar{c}$" to "$w_j \leq \hat{c}$ and (($j <$

NA and $\hat{c} > \bar{c}$) or $j < NB$)". In addition, we modified the last condition in the block 4 [use the dynamic programming lists] from "($\hat{c} < w_{NB-1}$ or $j \geq NB$) and ($\hat{c} < \bar{c}$)" to "($\hat{c} < w_{NB-1}$ or $j \geq NB$) and ($\hat{c} \leq \bar{c}$)". Second, in Pisinger's NRSUB algorithm ([4], page 160), if $s_t(c) \neq 0$ in the loop for variable t (lines 5–14), then an optimal solution was found. Therefore, we modified the loop to allow for leaving it as soon as possible.

To examine behaviour of the algorithms we used the *problems TODD*, *AVIS*, *P(E)* and *EVEN/ODD(E)* the detailed description of which are presented in the Martello & Toth [3]. All instances were solved on SPARCstation IPX on which the amount of available main memory and totally swap space are 32 and 64 megabytes, respectively. The reported running time is always expressed in seconds.

4.1 Problems TODD

The instances of problems TODD are given by

$$\begin{aligned} a_j &:= 2^{k+n+1} + 2^{k+j} + 1, \quad \text{with} \quad k = \lfloor \log_2 n \rfloor \\ c &:= \left\lfloor 0.5 \sum_{j=1}^{n} a_j \right\rfloor = (n+1)\,2^{k+n} - 2^k + \left\lfloor \frac{n}{2} \right\rfloor . \end{aligned}$$

Since these instances have the property $\left\lfloor \frac{1}{2} \sum_{j=1}^{n} a_j \right\rfloor = c$, the unmodified XS-algorithm can be applied directly to the PP with $c_j := a_j$, $j = 1, 2, \ldots, n$.

Table 1 gives the fastest running time among several trials for each n of this deterministic test problem. Since the weights of items grow exponentially the integer overflow occurs when $n \geq 23$.

Table 1: results of problems TODD

n	MTSL	NRSUB	XS
10	0.04	0.95	0.04
12	0.05	4.40	0.04
16	0.09	—	0.04
20	0.39	—	0.04
22	0.74	—	0.04

— Not enough memory.

It turns out that the preprocessing of the XS-algorithm is especially efficient for this type of instances. Explanation is simple. If n is odd, then the lower bound k of the solution length is greater than the upper bound $\lfloor n/2 \rfloor$. Consequently, no tree is constructed and only one subset is evaluated. If n is even, then $k = n/2$, and only one tree is constructed. Moreover, in this case, XS discards the solutions containing n. Therefore the subset $\{n-1, n-2, \ldots, \frac{n}{2}\}$ is evaluated first. Since its weight is less than $c(N)/2$, SKIP is performed, which results in termination of the search, because no '0' can be found in the corresponding bit pattern.

4.2 Problems AVIS

The instances of problems AVIS are given by

$$\begin{aligned} a_j &:= n(n+1)+j \\ c &:= \left\lfloor \frac{n-1}{2} \right\rfloor n(n+1) + \binom{n}{2}. \end{aligned}$$

Since such instances do not satisfy the condition $\left\lfloor \frac{1}{2}\sum_{j=1}^{n} a_j \right\rfloor = c$, we transform them to PP according to (9) and apply the modified XS-algorithm. Table 2 gives again the fastest running time among several trials for each n.

Table 2: results of problems AVIS

n	MTSL	NRSUB	XS
12	0.04	0.07	0.04
16	0.04	0.10	0.04
20	0.05	0.15	0.05
100	0.06	11.90	0.05
200	0.08	91.90	0.06
300	0.09	307.99	0.06
500	0.12	—	0.06
1000	0.20	—	0.06

— Not enough memory.

Similarly to problems TODD, the preprocessing of the XS-algorithm is extremely efficient for this type of instances. Again, explanation is simple. Since $c_{n+2} > c_{n+1}$ for all $n \geq 2$, XS evaluates only one subset when determining the lower bound k. Moreover, according to Avis ([8], page 1411)

$$\sum_{j \in J} a_j > c \quad \text{for all} \quad n \geq 2 \quad \text{and} \quad |J| \geq \left\lfloor \frac{n+1}{2} \right\rfloor$$

Therefore, $k = \lfloor \frac{n+1}{2} \rfloor$. Consequently, no tree is constructed when n is odd, and only one tree is constructed when n is even. In this tree, item $n+1$ is always selected, because $c_{n+2} > c_{n+1}$. Moreover, the weights of all solutions occurring during the search are less than $c(N')/2$, because the sum of c_{n+2} and $c(J)$ for each $J \subset N$ with $|J| = n/2$ is greater than $c(N')/2$. Therefore, the tree can be discarded.

4.3 Problems P(E)

In contrast to problems TODD and AVIS, the instances of problems P(E) are randomly generated. The weights a_j are constructed to be uniformly random in the interval $[1, 10^E]$ where E is a positive integer greater than 1, and $c := n \cdot 10^E/4$. Note that c is selected in such a manner, that about half the items can be expected to form optimal solutions. This implies that problems P(E) tend to be more difficult with growing E, because the

difficulty is related to the number of different solutions J with $a(J) = c$. As pointed out in Martello & Toth, truly difficult problems can be obtained only with very high values of 10^E. Table 3 and Table 4 give the results for problems P(3) and P(6), respectively. Each entry gives the average running time of 10 instances. In the case of P(6), the integer overflow occurs when $n > 2000$. The columns denoted XSL give the average running times for a variant of XS which uses *core problems* in similar way as MTSL does. The rationale of such modification is that, for large-size instances, too much time during the preprocessing stage is consumed for sorting. For example, in the case of $n = 100000$ in P(3), the total number of evaluated subsets by XS was only two or three. Thus almost all time was spent on preprocessing. Regarding the results for P(6), it should be pointed out that XS may become faster for larger values of n. We have observed that the average number of evaluated subsets was

$$\begin{aligned} 13\,457 &\quad \text{for} \quad n = 10, \\ 1\,301 &\quad \text{for} \quad n = 1000, \\ 200 &\quad \text{for} \quad n = 2000 \end{aligned}$$

In all these cases solutions J with $a(J) = c$ were found. Since the probability of hitting such a solution tends to increase with growing n, and since XS stops when it finds such a solution, it is obvious that the time spent on solving larger instances may be shorter.

Table 3: results of problems P(3)

n	MTSL	NRSUB	XS	XSL
12	0.054	0.114	0.045	0.048
16	0.048	0.105	0.048	0.050
20	0.052	0.105	0.048	0.052
100	0.054	0.095	0.051	0.052
1000	0.067	0.443	0.118	0.065
10000	0.182	5.679	0.932	0.185
100000	1.366	74.957	14.217	1.441

Table 4: results of problems P(6)

n	MTSL	NRSUB	XS	XSL
12	0.057	40.208	0.047	0.054
16	0.069	54.532	0.075	0.092
20	0.098	66.753	0.384	0.430
100	0.155	—	0.425	0.394
1000	0.152	—	0.165	0.495
2000	0.152	—	0.226	0.674

— Not enough memory.

4.4 Problems EVEN/ODD(E)

Similarly to problems P(E), the instances of problems EVEN/ODD(E) are randomly generated. They differ from instances of P(E) in the following aspects. The weights are again randomly distributed in the interval $[1, 10^E]$ but now they are required to be even numbers. The capacity is required to be an odd number given by $c := n \cdot 10^E/4 + 1$. Table 5 gives the average running times of 10 instances for $E = 3$.

Table 5: results of problems EVEN/ODD(3)

n	MTSL	NRSUB	XS
12	0.050	0.142	0.051
16	0.072	0.179	0.074
20	0.282	0.207	0.470
24	3.598	0.219	4.215
28	38.905	0.263	67.851
32	445.617	0.292	758.912

Note that the performance of the XS-algorithm is not as good as in the previous cases. We observed that the average of lower bounds for the length of solutions was 10.2 for $n = 32$. This indicates that relatively many trees must be dealt with in this problem type. However, we observed that the performance of XS is much less sensitive to changes in the "scaling factor" E, than the performance of MTSL and NRSUB. This is clearly demonstrated by comparison of Table 5 with Table 6. The latter gives the average running times of 10 instances for $E = 6$.

Table 6: results of problems EVEN/ODD(6)

n	MTSL	NRSUB	XS
12	0.057	53.694	0.051
16	0.095	72.802	0.086
20	0.119	77.912	0.394
24	0.329	100.624	4.265
28	2.703	117.710	63.782
32	26.728	130.994	733.481

5 Conclusion

We have presented a new algorithm for solving exactly the subset-sum problem and compared its experimental behaviour with two algorithms from recent literatures, namely the MTSL algorithm by Martello & Toth [3] and the NRSUB algorithm by Pisinger [4]. As one may expect, the proposed algorithm (called XS) does not always outperform MTSL and NRSUB. However, the computational experiments on the standard groups of instances have shown that XS has some promising features. It turns out that

- XS is economical in memory requirements, because it is based on the depth-first tree search and it does not use any component related to dynamic programming.
- XS is not too sensitive to the magnitude of input data, because the properties of the partition problem used in preprocessing are not influenced by the magnitude and range of weights, and because the basic strategy is based on a fixed order of items.
- XS is able to solve so called hard instances in reasonable time, because its orientation on narrowing the necessary search space as much as possible and discarding dominated states.

Since XS outperforms MTSL and NRSUB on problems TODD and AVIS, we may conclude that XS is good at solving instances the weights of which are distributed close to higher values. On the other hand the results for problems EVEN/ODD(E) suggests that XS is not good at solving instances with uniformly distributed weights. This indicates that XS is more sensitive to the distribution of weights than to the magnitude of weights. However our computational experiments are not sufficient for deriving some classification of distributions with respect to the performance of XS.

Acknowledgments

The authors are grateful to Dr. David Pisinger at DIKU in Denmark who kindly answered our question as to the algorithmic sketch in the paper and gave us a valuable suggestion.

References

[1] Richard M. Karp : "Reducibility among Combinatorial Problems", in R. E. Miller and J. W. Thatcher (eds.), *Complexity of Computer Computations*, Plenum Press, New York, pp.85–103 (1972).

[2] Michael R. Garey, David S. Johnson : *Computers and intractability : a guide to the theory of NP-completeness*, W. H. Freeman, San Francisco (1979).

[3] Silvano Martello, Paolo Toth : *Knapsack Problems: Algorithms and Computer Implementations*, John Wiley & Sons, Chichester, England (1990).

[4] David Pisinger : *Algorithms for Knapsack Problems*, Ph.D. thesis, Dept. of Computer Science, University of Copenhagen (DIKU), Denmark, February (1995).

[5] Stephen A. Vavasis : "Complexity Issues in Global Optimization: A Survey", in Reiner Horst and Panos M. Pardalos (eds.), *Handbook of Global Optimization*, Kluwer Academic Publishers, Dordrecht, Netherlands, pp.27–41 (1995).

[6] R. G. Jeroslow : "Trivial Integer Programs Unsolvable by Branch-and-Bound", *Mathematical Programming*, **6**, pp.105–109 (1974).

[7] Michael Todd : THEOREM 3. In V. Chvátal, "Hard knapsack problems", *Opns. Res.*, **28**, pp.1408–1409 (1980).

[8] David Avis : THEOREM 4. In V. Chvátal, "Hard knapsack problems", *Opns. Res.*, **28**, pp.1410–1411 (1980).

[9] S. N. N. Pandit and M. Ravi Kumar, "A Lexicographic Search for Strongly Correlated 0–1 Knapsack Problems", *Opsearch*, **30**, No.2, pp.97–116 (1993).

[10] Hiroshi IIDA : "An exact algorithm for the subset-sum problem", Master's thesis, School of Information Science at the JAIST, Hokuriku, February (1996).