Pergamon 0305-0548(94)00025-5

Computers Ops Res. Vol. 22, No. 3, pp. 321–334, 1995
Copyright © 1995 Elsevier Science Ltd
Printed in Great Britain. All rights reserved
0305-0548/95 $9.50+0.00

AN EFFICIENT PRUNING ALGORITHM FOR VALUE INDEPENDENT KNAPSACK PROBLEM USING A DAG STRUCTURE

CHA-HON SUN† and SHENG-DE WANG‡
Department of Electrical Engineering, EE Building, Rm. 441, National Taiwan University, Tapei 106, Taiwan, Republic of China

(Received October 1992; in revised form March 1994)

Scope and Purpose—The knapsack problem is a well known optimization problem. It has been proved to be NP-complete. Solving the knapsack problem can be viewed as a way to study some problems in number theory. There are some applications based on the knapsack problem, such as task scheduling and memory management problems in Operating Systems, crytography, Integer Linear Programming (ILP) problems, etc. The value independent knapsach problem (VIKP) is a special case of knapsack problem. Due to exponential explosion, the space consideration has been a bottleneck in this problem. In this paper, we use a data structure, directed acyclic graph (DAG), to store all the solutions of VIKP efficiently. We will show that the space usage using DAG is less than that of other data structures proposed in the past.

Abstract—In this paper, we propose an efficient pruning algorithm to solve the value independent knapsack problem. It stores all the solutions in a directed acyclic graph (DAG) using only $O(M\cdot n)$ space, where n is the problem size and M is the subset summation. Our algorithm is suitable for the case of $M \ll 2^n$. Also, we find a symmetric property that can improve many heuristic algorithms proposed in the past.

1. INTRODUCTION

Given a positive integer multiset A with cardinality n and a nonnegative integer M, we are required to find all the subsets, called solution sets, of A such that these subset sums are equal to M. The problem is the so called "value independent knapsack problem (VIKP)." Some also call the problem "Stickstacking Problem." The most important applications are cargo loading, cutting stock and job scheduling. A major difference between our algorithm and others proposed in the past is that our algorithm finds "all" rather than "one" of the solutions. The VIKP has been proven to be NP-complete in Ref. [1]. An n-element set has exact 2^n possible subsets. Many heuristic algorithms [1–3, 6, 7] have been proposed to solve the problem. However, these algorithms have exponential complexities both in time and space. Bruce [8] proposed a partitioning algorithm to solve the number of the solutions using recursive technique. It is not further discussed how to find these solutions in Ref. [8]. Kolesar [2] proposed a branch and bound algorithm which has the complexities of time $T=O(2^n)$ and space $S=O(2^n)$. Horowitz and Sahni [1] proposed a two-list algorithm which has the following complexities: $T=O(\max(2^{n/2}, Q\cdot n))$, $S=O(2^{n/2})$, where Q is the number of solution sets. Ahrens and Finke [7] independently proposed a dynamic programming approach similar to Ref. [1]. The linked list data structure is used in Refs [1, 2, 8]. A two-list four-table algorithm using the heap structure is proposed by Schroeppel and Sharmir [3]. The complexities are $T=O(2^{n/2})$ and $S=O(2^{n/4})$. In Ref. [7], Ahrens and Finke proposed a similar four-table algorithm which uses tournament sorting technique. Some approximation approaches are derived to find feasible solution

†Cha-Hon Sun is a Ph.D. candidate at the Department of electrical engineering, National Taiwan University, Taipei, Taiwan. His research interests include parallel and distributed algorithms, cryptography, and reliability analysis.
‡Sheng-De Wang received a B.S. degree from National Tsing Hua University, Hsinchu, Taiwan, and M.S. and Ph.D. degrees in electrical engineering from National Taiwan University, Taipei, Taiwan. He is a professor in the department of electrical engineering at National Taiwan University, Taipei, Taiwan. His research interests include artificial intelligence, parallel processing, and neuro-computing.

instead of optimal one. Ibrarra and Kim [10] proposed an approximation algorithm to find a feasible solution in the 0–1 Knapsack Problem.

Because of NP-completeness, many parallel algorithms were also proposed to get high speedup or reduce storage requirement. Karnin [4] improved the two-list four-table algorithm on a multiprocessor system with $2^{n/6}$ processors to yield $\mathrm{T}=O(2^{n/2})$ and $\mathrm{S}=O(2^{n/6})$. Ferreira [5] proposed a one-list algorithm which has the following complexities: $\mathrm{P}=O((2^{n/2})^{1-\varepsilon})$, $\mathrm{T}=O((n\cdot 2^{n/2})^{\varepsilon})$ and $\mathrm{S}=O(2^{n/2})$, where P is the processor number and $0\leqslant\varepsilon<1$. As for our algorithm, the complexities are as follows: $\mathrm{T}=O(\min(Mn^2, Qn)+Mn/L)$ and $\mathrm{S}=O(\min(Mn, Q\cdot n)+Mn)$, where L is the bit length of one memory cell. It should be pointed out that our algorithm is suitable in the case of $M\ll 2^n$. In fact, small storage requirement makes our algorithm excellent and practical in many applications.

We organize the rest of this paper as follows. In Section 2, we make some notation defintions and introduce the two-list algorithm [1, 7] and a bit-shifting technique briefly. In Section 3, we introduce our algorithm using five Subsections. In Subsection 3.1, a symmetric property related with knapsack problem is examined. In Subsection 3.2, we introduce a partition approach. In Subsection 3.3, an innovative algorithm, which saves the solution sets in a directed tree structure is presented. In Subsection 3.4, a modified algorithm is proposed to reduce the computation time and space without loss of any information. A DAG is built instead of a directed tree. In Subsection 3.5, we give the final algorithm to further reduce the computation time. In Section 4, we show the experimental results and discuss these results briefly. Section 5 is a further discussion on our algorithm. At last we make some conclusions in Section 6.

2. DEFINITION AND PRELIMINARY

2.1. Notation definition

The following notation will be used in this paper.

A_v,	a set, $\{a_1, a_2, \ldots, a_v\}$, that contains the first v elements of A. Thus, for an n-element set, the notations A_n and A can be used interchangeably;
$\lvert A_v\rvert$,	the number of elements in set A_v, i.e. $\lvert A_v\rvert=v$;
$\mathrm{SUM}(A_v)$,	the summation of all elements in A_v. i.e. $\sum_{i=1}^{v} a_i$;
$\langle A_v, m\rangle$,	a VIKP with multiset A_v and positive integer m;
$\mathrm{MAX}(A_v)$,	the value of the maximal element in A_v;
$\mathrm{MIN}(A_v)$,	the value of the minimal element in A_v;
B,	$\{B_1, B_2, \ldots, B_Q\}$, the set of the solution sets of $\langle A_n, M\rangle$, where $\mathrm{SUM}(B_i)=M$, $\forall B_i\in B$ and $\lvert B\rvert=Q$;
$\bar{B}$,	$\{\bar{B}_1, \bar{B}_2, \ldots, \bar{B}_Q\}$, the complement of set B, where $\bar{B}_i$ is the complement of B_i under A_n, i.e. $\bar{B}_i=A_n-B_i$.
$f(A_v, m)$,	the number of the solution sets given $\langle A_v, m\rangle$. Thus, $f(A_n, M)=\lvert B\rvert=Q$.

The VIKP, $\langle A_n, M\rangle$, can be stated mathematically as follows.

$$\text{Find}\quad V=(v_1, v_2, \ldots, v_n) \tag{1}$$

$$\text{subject to}\quad \sum_{i=1}^{n} a_i v_i=M, \tag{2}$$

$$v_i=0 \text{ or } 1, (i=1, 2, \ldots, n). \tag{3}$$

Bruce [8] and this paper use the above definition. Some other papers use another definition as follows.

$$\text{Find}\quad V=(v_1, v_2, \ldots, v_n) \tag{4}$$

$$\text{subject to}\quad \sum_{i=1}^{n} a_i v_i\leqslant S, \tag{5}$$

$$v_i=0 \text{ or } 1, (i=1, 2, \ldots, n). \tag{6}$$

Assume M is the solution of equation (4). In the case of $2^2\gg S$, finding M is not the dominant factor in our algorithm. Thus, we use equations (1), (2), (3) instead of equations (4), (5), (6).

2.2. *Literature for knapsack problem*

Kolesar [1] proposed a branch-and-bound algorithm, called *Knapsack*(1*a*) in this paper, as described in the following

Algorithm Knapsack(1*a*)
```
        step 1:  Initialize W={(0,0)}; IC=1;
        step 2:  do i=1,n
                 {  if ($a_i$+P<M) then W=W∪{W+($a_i$,IC)}
                    if ($a_i$+P=M) then output (IC+V);
                    IC=IC+IC;
                 }
```

In Algorithm *Knapsack*(1a), W is a set of 2-tuples (P, V), where P is a partial sum and V is a binary encoding vector of n bits, $(v_1, v_2, \ldots, v_n)$, such that $\sum_{j=1}^{n} a_j v_j = P$.

Example 1: Given $\langle\{1, 3, 4, 5\}, 8\rangle$, we have the following steps.

```
Initialize W={(0, 0000)}
           IC=0001 ;
i=1        W={(0, 0000), (1, 0001)}
           IC=0010 ;
i=2        W={(0, 0000), (1, 001), (3, 0010), (4, 0011)}
           IC=0100 ;
i=3        W={(0, 0000), (1, 0001), (3, 0010), (4, 0011), (4, 0100), (5, 0101),
              (7, 0110), (8, 0111)}
           IC=1000 ;
i=4        W={(0, 0000), (1, 0001), (3, 0010), (4, 0011), (4, 0100), (5, 0101),
              (7, 0110), (8, 0111), (5, 1000), (6, 1001), (8, 1010)}
           IC=10000 ;
```

After the step i=4, the solution sets {4, 3, 1} and {5, 3} whose V are 0111 and 1010 respectively are found. In the example, there are 11 elements in W after the step i=4.

If we use exhaustive search, there will be 16 elements in set W finally. *Knapsack*(1*a*) is superior to the exhaustive search because of the heuristic in step 2. In step 2 of *Knapsack*(1*a*), it reduces the size of W by pruning the useless subsets whose sums, P, are greater than M.

Horowitz and Sahni [2] and Ahrens and Finke [7] independently proposed a two-list algorithm by modifying *Knapsack*(1*a*). In this paper, we call the two-list algorithm as *Knapsack*(1*b*). *Knapsack*(1*b*) splits $\langle A_n, M\rangle$ into $\langle U_{n/2}, M\rangle$ and $\langle V_{n/2}, M\rangle$ where $U_{n/2}=A_{n/2}$ and $V_{n/2}=\overline{U_{n/2}}$ under A_n. It first applies *Knapsack*(1*a*) to get sets W_U and W_V for the subproblems $\langle U_{n/2}, M\rangle$ and $\langle V_{n/2}, M\rangle$, respectively. Then, W_U and W_V are merged to get all the solution sets.

In Refs [3] and [7], a four-table algorithm is proposed. The major difference between Refs [3] and [7] is that a heap is used in Ref. [3]; while arrays are used to implement the tournament sort in Ref. [7]. We only describe the algorithm of Ref. [7] in this paper. The four-table algorithm, called *Knapsack*(1*c*), has a similar splitting approach to *Knapsack*(1*b*). *Knapsack*(1*c*) firstly splits A_n into four distinct sets, R, S, T, U. Secondly, $\langle R, M\rangle$, $\langle S, M\rangle$, $\langle T, M\rangle$, and $\langle U, M\rangle$ are solved, like *Knapsack*(1*a*), in four tables. Finally, the solutions are got by merging the four tables using tournament sorting technique.

2.3. *Bit-shifting technique*

To efficiently store the subset sums, we present a bit-shifting technique that makes use of two bit vectors. The steps of the technique are as follows.

The bit-shifting procedure
```
        Initialize: $T[0]=Backup[0]=1$, $T[i]=Backup[i]=0, i=1,2,\ldots,M$;
        DO i=1,n
        {
          $Backup=Backup \ll a_i$ bits ; /* '≪': shift-left operator */
          $T=T|Backup$ ; /* '|': logical OR operator */
        if $T[M] = 1$
          then $a_i$ is an element of some solution sets;
        $T[M] = 0$ ;
          $Backup = T$ ;
      }
```

In the above procedure, T and *Backup* are two bit vectors of size $M + 1$. To illustrate the bit-shifting technique, we use the same example as mentioned above.

Example 2: Given $\langle\{1, 3, 4, 5\}, 8\rangle$, we have the following steps:

		8	7	6	5	4	3	2	1	0
Initialize	T	0	0	0	0	0	0	0	0	1
i = 1	T	0	0	0	0	0	0	0	1	1
i = 2	T	0	0	0	0	1	1	0	1	1
i = 3	T	1	1	0	1	1	1	0	1	1

In the steps i = 3, a "1" in $T[8]$ implies that $a_3 = 4$ is an element of some solution sets.

i = 4	T	1	1	1	1	1	1	0	1	1

Again, this step shows $a_4 = 5$ is also an element of some solution sets. Note, in the step i = 4, that $T[2]$ = "0" means no solution if $M = 2$ in this example. It implies that we can find M in equations (4–6) if $T[S]$ = "0". The time complexity of the bit-shifting procedure is $O(n \cdot M/L)$.

The parallel implementation of the bit-shifting technique on an SIMD multiprocessor system can easily be obtained. If one memory cell has the length of L bits, it needs $\lceil M/L \rceil$ memory cells to implement for a bit vector. We need a tightly coupled synchronous multiprocessor with $\lceil M/L \rceil$ processors to share $\lceil M/L \rceil$ memory cells. Assume each processor can access arbitrary memory cells without conflict at the same time. It is the well known EREW (Exclusive Read Exclusive Write) shared memory model. For instance, processors P_x and P_y can access memory locations M_x and M_y, respectively, at the same time if $M_x \neq M_y$. Consider that we wish to shift array T in left direction for D bits. Let $Q = \lfloor D/L \rfloor$, R = D modulo L. Clearly, D bits = Q cells + R bits. We analyze arbitrary memory cells, say T[X], which shifts L bits in left direction. We find that the most significant (L–R) bits of T[X] will be moved to the least significant (L–R) bits of T[X + Q + 1], and the least significant R bits of T[X] will be moved to the most significant R bits of T[X + Q]. For example, if D = 38, L = 16 then Q = 2, R = 6. Assume X = 2 and T[2] = 1111111111000000. When it finishes the shift-left operation, T[X + Q + 1] = T[5] will become xxxxxx1111111111 and T[4] become 000000xxxxxxxxxx, where "x" represents "unknown."

The parallel bit-shifting technique will be run in synchronous mode. Each processor is assumed to know its own processor identification number in advance and execute a procedure shown as follows.

Parallel bit-shifting procedure / For processor X */*

```
do i = 1,n
step 1: read A[i] to determine shifting amount.
step 2: R = A[i] mod L; Q = ⌊A[i]/L⌋;
step 3: tmpL=T[X]>>R; tmpR=T[X]<<(L-R);
step 4: T[X+Q+1]=T[X+Q+1] | tmpL;
        /* if X+Q+1>H then do noting just to
           delay one operation cycle for synchronization,
        where $H = \frac{M+1}{L}$ */
step 5: T[X+Q]=T[X+Q] | tmpR;
step 6: if (X=H)
        {
          r=M modulo L;
          Pattern=1;
          Pattern=Pattern<<r;
          find_out=T[H] & Pattern;
        /* detect whether solution is found */
        /* & denotes logic AND operation */
          if(find_out)
          {
            generate <A_i, M-A[i]> ;
            T[H]=T[H] &~pattern ;
          }
        }
        else
        {
          delay one operation cycle for synchronization;
        }
enddo
```

In the above procedure, tmpL and tmpR are local variables for each processor.

3. OUR ALGORITHM

3.1. Symmetric property

For an aribitrary multiset A with cardinality n, there are exact 2^n subsets of set A. The maximal sum of subset is SUM(A) and the minimal sum of subset is 0. Figure 1 shows that all the partial sums lie in the range of $[0..\mathrm{SUM}(A_n)]$, where x-axis represents the sum, m, of subset and y-axis represents the number of solution sets for a specific m.

Lemma 1. $\sum_{m=0}^{SUM(A_n)} f(A_n, m) = 2^n$.

Proof: The result is trivial because a multiset with cardinality n has 2^n subsets and these subset sums are in the range of $[0...\mathrm{SUM}(A_n)]$. □

Given $\langle A_n, M\rangle$, *Knapsack*(1*a*) saves those subsets whose summations are less than or equal to M. Therefore, we can write the space complexity in exact bound $\Theta(\sum_{m=0}^{M} f(A_n, m))$ instead of asymptotic upper bound $O(2^n)$. The space complexity is $\Theta(\sum_{m=0}^{M} (f(U_{n/2}, m) + f(V_{n/2}, m)))$ for *Knapsack*(2*b*).

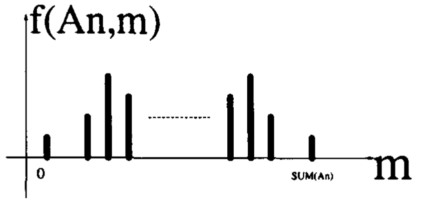

Fig. 1. All subset sums of A_n drop in $[0 \ldots \mathrm{SUM}(A_n)]$.

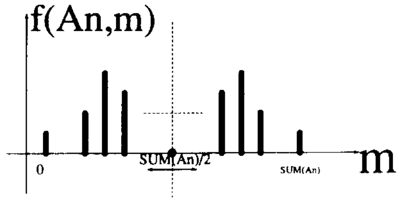

Fig. 2. Symmetry with m = SUM(A_n)/2.

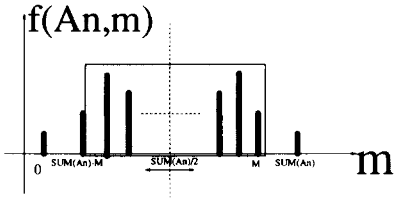

Fig. 3. Shadow area is the improvement of space and time for m = M.

Lemma 2. $f(A_n, M) = f(A_n, \mathrm{SUM}(A_n) - M)$ (shown in Fig. 2).

Proof: Assume $B = \{B_1, B_2, \ldots, B_Q\}$ is the set of solution sets for the problem $\langle A_n, M\rangle$. We know $\mathrm{SUM}(B_1) = M$. It implies $\mathrm{SUM}(\bar{B}_1) = \mathrm{SUM}(A_n) - \mathrm{SUM}(B_1) = \mathrm{SUM}(A_n) - M$. Based on this concept, $\bar{B} = \{\bar{B}_1, \bar{B}_2, \ldots, \bar{B}_Q\}$ is the set of solution sets for the problem $\langle A_n, \mathrm{SUM}(A_n) - M\rangle$.

It is apparent that from Lemma 2 we have the following theorem.

Theorem 1. *To find all solutions of a knapsack problem $\langle A_n, M\rangle$ is equivalent to find the complement of all solutions of $\langle A_n, SUM(A_n) - M\rangle$.*

For those algorithms based on the branch and bound approach, like Refs [1–3, 6, 7], the computation time and space increase when the value M increases. For example, given $\langle A_n, m\rangle$, the *Knapsack*(1*a*) needs $\Theta(n)$ in time and $\Theta(1)$ in space when m = MIN(A_n). Nonetheless, it spends $\Theta(2^n)$ in both time and space when m = SUM(A_n) − MIN(A_n). In this situation, Theorem 1 can be applied to reduce the computation time and space. Cite *Knapsack*(1*a*) as an example by applying Theorem 1. When $M > \mathrm{SUM}(A_n)/2$, we can remove those elements (P, V) satisfying $P > \mathrm{SUM}(A_n) - M$ from set W, and thus $\sum_{m=\mathrm{SUM}(A_n)-M+1}^{M} f(A_n, m)$ elements will be removed in total. Figure 3 depicts this idea. In Fig. 3, the shadow area is the improvement in execution time and space when we apply the symmetric property.

Let's solve *Example 1* using Theorem 1. According to Theorem 1, we can solve the equivalent problem, $\langle\{1, 3, 4, 5\}, 5\rangle$, because of SUM({1, 3, 4, 5}) = 13 and $M = 8$. There are only 7 elements in set W finally. Four elements, (7, 0110), (8, 0111), (6, 1001) and 8, 1010), are removed from the set W in *Example 1*. The solution sets of the equivalent problem are (5, 1000) and (5, 0101). Therefore, the solution sets of the original problem are (8, 0111) and (8, 1010), respectively, which are the same as the solution sets in *Example 1*.

In fact, the maximal computation time and space are $\Theta(2^{n-1})$ which happen on m = SUM(A_n)/2. And, the farther the distance between M and SUM(A_n)/2 is, the less the computation time and space are needed. The complexities of time and space were analyzed and represented with asymptotic upper bounds in Refs [1–7, 9]. In this paper, we will use exact bounds to depict the complexities instead of asymptotic upper bounds.

3.2. *Partition approach*

A partition approach is introduced to divide set B into n distince sets $S_1, S_2, \ldots, S_n$ as follows.

The partition approach

```
Assume B = $\{B_1, B_2, \ldots, B_Q\}$ is the set of solution sets of $\langle A_n, M\rangle$
Do i = n down to 1
{
    $S_i = B_j \mid B_j \in B, a_i \in B_j\}$
    $x_i$ = cardinality of $S_i$ /* $x_i = |S_i|$ */
    $B = B - S_i$
}
```

For the first time, pick out all the elements $B_i's$ that contain element a_n from set B. Gather these $B_i's$ and form a new set of sets S_n. Let $x_n = |S_n|$. After this step, $|B| = Q - x_n$. Repeating the steps n times, we can get sets $S_n, S_{n-1}, \ldots, S_1$, and their cardinalities are $x_n, x_{n-1}, \ldots, x_1$, respectively.

Theorem 2. *After applying the above partition approach, it can be shown that the following properties hold.*

1. $f(A_n, M) = x_1 + x_2 + \cdots + x_n = Q$.
2. $S_i' = \{y - \{a_i\} \mid y \in S_i\}$ is the set of solution sets of $\langle A_{i-1}, M - a_i \rangle$
3. $f(A_{n-v}, M) = \sum_{i=1}^{n-v} x_i$

Proof:

(1) According to the partition approach, we have $B = \bigcup_{i=1}^{n} S_i$ and $S_i \cap S_j = \Phi$, $i \neq j$. Thus, complete the proof.

(2)(3) Because property 2 and 3 are very intuitive, we ignore the proof. □

Using property 1 and property 2 of Theorem 2, we can divide $\langle A_n, M \rangle$ into n subproblems. Each subproblem is also a value independent knapsack problem. This constructs a recursive form. There are three boundary conditions which can return from the recursion.

(1) BC 1: $f(A_i, 0) = 1$.
(2) BC 2: $f(\{\}, m) = 0$, $m \neq 0$.
(3) BC 3: $f(A_i, z) = 0$, $z < 0$.

Any subproblem will eventually be reduced into one of these three conditions. Bruce [8] derived a recursive formula, which is similar to Theorem 2 of this paper, to calculate the number of the solutions. However, it is not further discussed how to find the solutions in Ref. [8]. Our algorithm extends the recursive formula to find all the solutions.

3.3. An efficient pruning algorithm [Knapsack(2a)]

A directed tree structure is built based on the partition approach and the property 2 of Theorem 2. In Fig. 4, each node is labeled as a 2-tuples $\langle A_i, m \rangle$ and each edge incident to node $\langle A_i, m \rangle$ is labeled as a_{i+1}.

Applying the partition approach, we classify S_i, $i = 1, 2, \ldots, n$ into two calsses. The first class is $|S_i| = 0$ and the second class is $|S_i| \neq 0$. If $\langle A_i, m \rangle$ is in the first class, it implies it will be reduced to BC2 or BC3 eventually. Our algorithm, *Knapsack*(2*a*), will not generate those nodes in the first class by applying the bit-shifting technique. The *Knapsack*(2*a*) is as follows.

Algorithm Knapsack(2*a*)
```
input:  <An,M>
output: solution tree.
begin
  if(M=0) then return
  create all the child nodes of the second class by bit-shifting technique.
  for every child node
    call Knapsack(2a) and transfer the child node as input.
end
```

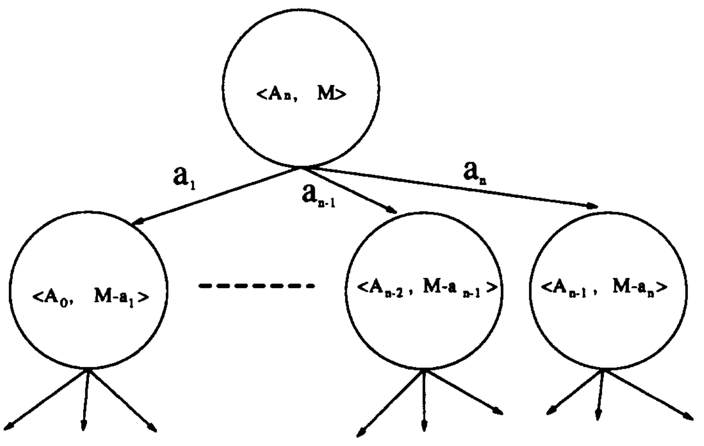
$\langle A_n, M \rangle$
a_1
a_{n-1}
a_n
$\langle A_0, M-a_1 \rangle$
$\langle A_{n-2}, M-a_{n-1} \rangle$
$\langle A_{n-1}, M-a_n \rangle$

Fig. 4. A directed tree based on the partition technique.

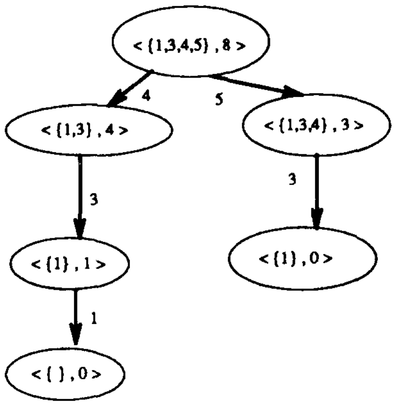

Fig. 5. $\{\{1, 3, 4\}, \{3, 5\}\}$ is the solution set of $\langle\{1, 3, 4, 5\}, 8\rangle$.

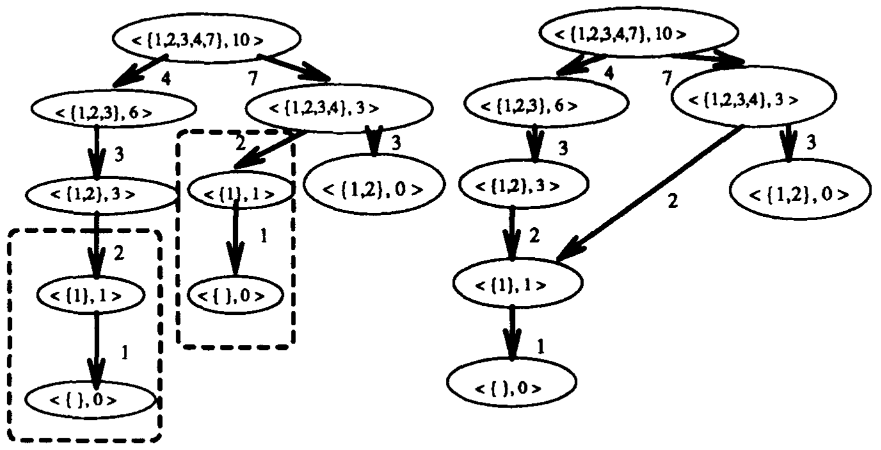

Fig. 6. Transform a directed tree into a DAG by removing redundant nodes.

Figure 5 demonstrates the solution tree of *Example 2*, $\langle\{1, 3, 4, 5\}, 8\rangle$. At first, node $\langle\{1, 3, 4, 5\}, 8\rangle$ generates two children $\langle\{1, 3\}, 4\rangle$ and $\langle\{1, 3, 4\}, 3\rangle$ after the step i = 3 and i = 4 in *Example 2*, respectively. It implies that $\langle\{1\}, 5\rangle$ and $\langle\{\ \}, 7\rangle$ have no solution. When we complete the solution tree, we can find each leaf of the tree satisfying BC 1.

We claim two facts:

(1) That the solution tree of $\langle A_n, M\rangle$ has Q leaf nodes means there are Q solution sets;

(2) Every path from root to any leaf constructs a solution set by gathering the arc labels traversed. The time and space complexities of *Knapsack*(2*a*) are $\Theta(f_{t2a}\cdot Q)$ and $\Theta(f_{s2a}\cdot Q)$, respectively, where $f_{t2a}=O(n^2M/L)$ and $f_{s2a}=O(n)$.

3.4. *Modified pruning algorithm (Knapsack(2b))*

As mentioned above, the computation time and space are linearly proportional to Q. However, Q varies from problem to problem. The mean value of Q is $Q=2^n/\mathrm{SUM}(A_n)$. For large n, Q is still of exponential bound. We observe the solution tree of $\langle A_n, M\rangle$. Each leaf node is one of the n labels $\langle A_i, 0\rangle$, $i=0, 1, 2, \ldots, n-1$. By pigeonhole principle, there exist some leaf nodes which have the same label when $Q>n$. So do the internal nodes when $Q>M*n$. The pigeonhole principle describes a fact that there must exist one pigeonhole containing more than one egg if we lay n + 1 eggs in n pigeonholes. The major improvement of *Knapsack*(2*b*) is to remove the redundant nodes. Applying *Knapsack*(2*b*), the directed tree will turn into a DAG. Figure 6 demonstrates how to transform a directed tree into a DAG. In Fig. 6, dashed rectangulars are of the same. Note that an internal node is redundant, its descendant nodes are, too.

Algorithm *Knapsack*(2*b*) follows *Knapsack*(2*a*) except that it checks whether the certified node is required to be created or not before actually creating this node. If the node has existed, we cite the node directly. Otherwise, we generate this node. The time and space complexities of *Knapsack*(2*b*) are $O(f_{t2b}\cdot\min(M, Q))$ and $O(f_{s2b}\cdot\min(M, Q))$, respectively, where $f_{t2b}=O(Mn^2)$ and $f_{s2b}=O(n)$. The space requirement in *Knapsack*(2*b*) is only proportional to $\min(M\cdot n, Q\cdot n)$, which is a significant improvement over those known algorithms.

We can further improve the performance by reordering the set A_n in increasing order. The partition approach divides $\langle A_n, M\rangle$ into n subproblems of the form $\langle A_i, M-a_{i+1}\rangle$ using the bit-shifting technique. If $a_n=\text{MAX}(A_n)$, $M-a_n$ is smaller than $M-a_i$ for any $a_i\in A_n$ and the subproblem, $\langle A_{n-1}, M-a_n\rangle$, will be computed faster by using the bit-shifting technique, because it is linearly proportional to M. It prunes off not only the node itself but also its descendant nodes.

3.5. *The final algorithm (Knapsack(2c))*

We find that there still exists redundant operations in *Knapsack*(2*b*). The *Knapsack*(2*b*) recursively divides the problems into subproblems until BC 1. The bit-shifting procedure is invoked repeatedly by every subproblem. Each subproblem can be viewed as a partial replay of the original problem because every subproblem set is a subset of the original problem set. Therefore, it is redundant if every subproblem repeatedly invokes bit-shifting procedure. To avoid the redundancy, we use additional n bit vectors, $T_1, T_2, \ldots, T_n$, with the length of $M+1$. In the beginning, the bit-shifting procedure scans the original problem set and uses T_i to stored the information with respect to $\langle A_i, M\rangle$, $i=1, 2, \ldots, n$. In *Example 2*, the value contained in T in the step $i=1$ is stored in T_1 and that in the step $i=2$ is stored in T_2, and so forth. Then, for each subproblem, we use table-look-up method to get information from the n vectors instead of invoking the bit-shifting procedure. If we wish to solve the subproblem $\langle A_i, m\rangle$. According to the partition approach, $\langle A_j, m-a_{j+1}\rangle$, $j=0, 1, \ldots, i-1$, are the subproblems of $\langle A_i, m\rangle$. We must determine which nodes need to be generated. We perform the table-look-up method as follows. Check whether $T_j[m]$ is equal to "1" or not, for $j=1, 2, \ldots, i$. If $T_j[m]$ is equal to "1," the node $\langle A_{j-1}, m-a_j\rangle$ must be generated. In this way, no more redundant operation occurs.

4. EXPERIMENTAL RESULTS AND DISCUSSION

To fairly compare our algorithm with other algorithms, we adopt three types of data set given in Ref. [1] as our experimental data. Algorithms, *Knapsack*(1*a*), (1*b*), (1*c*), (2*a*), (2*b*), (2*c*), are run respectively using SUN-workstation. The experimental results are listed according to the computation time and space used, respectively. We list the complexities of these algorithms on both time and space in Table 1.

The first type of data set consists of sequential numbers, i.e. $a_i=i$. Four different M's are used for fair comparison. They are: (1) $M_1=\text{MAX}(A_n)$; (2) $M_2=\text{SUM}(A_n)/3$; (3) $M_3=\text{SUM}(A_n)/2$; and (4) $M_4=2\text{SUM}(A_n)/3$. Tables 2 and 3 are the results of the computation time and the amount of memory used, respectively. The objective of comparing M_2 and M_4 is to verify the improvement due to the symmetric property. Note that *Knapsack*(1*a*), (1*b*) and (1*c*) need to store all combinations of subsets. For example, the VIKP with $n=25$, $M=\text{SUM}(A_n)/2$ in Table 3, has totally 353,743 solutions. According to Tables 2 and 3, *Knapsack*(2*b*) is far superior to *Knapsack*(2*a*) both in time and space for any M_i. It implies that there are many redundancies in *Knapsack*(2*a*). All algorithms

Table 1. Complexity comparison of various algorithms

	Time	Space
(1a)	$\Theta(\sum_{i=0}^{M} f(A_n, i))$	$\Theta(\sum_{i=0}^{M} f(A_n, i))$
(1b)	$\Theta(\sum_{i=0}^{M} f(U_{n/2}, i)+\sum_{j=0}^{M} f(V_{n/2}, j)+Q)$	$\Theta(\sum_{i=0}^{M} f(U_{n/2}, i)+\sum_{j=0}^{M} f(V_{n/2}, j)+Q)$
(1c)	$\Theta(\sum_{i=0}^{M} f(R, i)+\sum_{j=0}^{M} f(S, j)+\sum_{k=0}^{M} f(T, k)+\sum_{l=0}^{M} f(U, l)+Q)$	$\Theta(\sum_{i=0}^{M} f(R, i)+\sum_{j=0}^{M} f(S, j)+\sum_{k=0}^{M} f(T, k)+\sum_{l=0}^{M} f(U, l)+Q)$
(2a)	$O(Q\cdot n^2/M)$	$O(\max(nQ, M))$
(2b)	$O(\min(M^2n^2, Mn^2Q))$	$O(\min(Mn, Qn))$
(2v)	$O(\min(Mn^2, Q\cdot n)+M\cdot n)$	$O(\min(Mn, Q\cdot n)+M\cdot n)$

where $Q=f(A_n, M)$ and $R\cup S\cup T\cup U=A_n$.

Table 2. Sequential number (time in milliseconds)

M	*n*	1a	1b	1c	2a	2b	2c
A	20	16.7	16.7	< 16.7	< 16.7	< 16.7	16.7
	25	50.0	33.3	16.7	< 16.7	< 16.7	16.7
	30	116.7	50.0	16.7	33.3	< 16.7	16.7
	35	350.0	116.7	33.3	66.7	33.3	33.3
	40	783.3	266.7	66.7	133.3	50.0	50.0
	50	3633.2	1233.3	200.0	483.3	66.7	83.3
	60	13,949.4	4633.1	766.6	1416.6	166.7	166.7
	99	—	—	—	—	916.6	933.3
B	20	2983.2	150.0	333.0	1150.0	83.3	66.7
	25	—	2416.6	4049.8	19,899.2	233.3	216.7
	30		—	—	—	650.0	483.3
	35					1483.3	1083.3
	40					3099.9	1899.9
	50					10,932.9	5516.4
	80					174,493.0	55,931.1
C	20	14,066.1	300.0	633.3	2583.2	100.0	83.3
	25	—	6733.1	10,416.2	—	316.7	216.7
	30		—	—		883.3	516.7
	35					2066.6	1016.6
	50					16.132.7	5233.1
	80					271,072.5	50,881.3
D	20	25,249.0	166.7	450.0	1150.0	83.3	66.7
	25	—	2516.6	4616.5	19,899.2	233.3	216.7
	30		—	—	—	650.0	483.3
	35					1483.3	1083.3
	40					3099.9	1899.9
	50					10.932.9	5516.4
	80					174,493.0	55,931.1

—, Stop due to > 20 MBytes.
A, $M = \max(A_n)$.
B, $M = \mathrm{SUM}(A_n)/3$.
C, $M = \mathrm{SUM}(A_n)/2$.
D, $M = 2\mathrm{SUM}(A_n)/3$.

Table 3. Sequential number (space in KBytes)

M	*n*	1a	1b	1c	2a	2b	2c
A	20	5.8	4.9	2.2	2.9	2.0	2.1
	25	14.1	11.5	4.4	6.5	3.5	3.5
	30	31.8	27.9	8.7	13.8	5.4	5.6
	35	67.3	58.9	16.7	27.8	8.1	8.3
	40	135.9	124.9	37.6	53.3	11.6	11.8
	50	493.5	469.4	125.8	178.1	21.3	21.6
	60	1593.5	1553.2	477.4	536.3	35.5	35.9
	99	—	—	—	—	152.2	153.4
B	20	1658.5	102.7	136.1	367.6	17.2	17.3
	25	—	1593.9	2483.3	6903.3	43.8	44.1
	30		—	—	—	95.2	95.7
	35					182.9	183.9
	40					321.6	322.9
	50					843.0	844.6
	80					5913.5	5924.1
C	20	8311.1	208.7	300.3	887.4	20.5	20.7
	25	—	4320.2	6914.3	—	49.5	50.0
	30		—	—		103.7	104.6
	35					193.3	194.6
	50					825.8	829.7
	90					5530.1	5607.5
D	20	14,832.9	112.4	136.2	367.6	17.2	17.3
	25	—	1736.8	2580.1	6903.3	43.8	44.1
	30		—	—	—	95.2	95.7
	35					182.9	183.9
	40					321.6	322.9
	50					843.0	844.6
	80					5913.5	5924.1

—, Stop due to > 20 MBytes.
A, $M = \mathrm{MAX}(A_n)$.
B, $M = \mathrm{SUM}(A_n)/3$.
C, $M = \mathrm{SUM}(A_n)/2$.
D, $M = 2\mathrm{SUM}(A_n)/3$.

but *Knapsack*(2*b*) and (2*c*) are terminated due to lack of memory (20 Mtytes) when $n > 30$. Consequently, *Knapsack*(2*b*) is the most superior in space and *Knapsack*(2*c*) is the most superior in time among the five algorithms in this type of data set.

The second type of data set comprises random numbers in the range [1 .. 100]. We run the algorithms using the following M's: (1) $M_1 = 100$; (2) $M_2 = \mathrm{SUM}(A_n)/3$; (3) $M_3 = \mathrm{SUM}(A_n)/2$; (4) $M_4 = 2\mathrm{SUM}(A_n)/3$; and (5) $M_5 = \mathrm{SUM}(A_n) + 1$. Tables 4 and 5 list the results of time and space requirements, respectively. In this test, we sort the data set in increasing order before running *Knapsack*(2*c*). In fact, data order is also influential to *Knapsack*(1*b*). If there are t partial sums in $T_{n/2}$ and u partial sums in $U_{n/2}$, it implies there are $t \cdot u = Q$ solution sets, where Q is a constant. Because t and u represent the space used by $T_{n/2}$ and $U_{n/2}$, respectively, the best case is $u = t = \sqrt{Q}$. It uses space $2\sqrt{Q}$ in total. The worst case is $u = 1$ and $t = Q$ as it uses total space Q. This implies we can reduce memory usage by feasibly splitting these two subsets $U_{n/2}$ and $T_{n/2}$, but we have no idea about how to split it. $M_5 = \mathrm{SUM}(A_n) + 1$ is run for the case when the problem has no solution. Our algorithms *Knapsack*(2*a*) and *Knapsack*(2*b*) can find the value M such that $\langle A_n, M \rangle$ has no solution in time complexity only $O(M \cdot n)$ if such an M exists. For convenience, we set $M = \mathrm{SUM}(A_n) + 1$ to represent that the problem has no solution. Although *Knapsack*(1*a*) and *Knapsack*(1*b*) can check whether $M > \mathrm{SUM}(A_n)$ or not, they can't check whether such cases as $M < \mathrm{SUM}(A_n)$ have solutions in advance. For this reason, we assume *Knapsack*(1*a*) and *Knapsack*(1*b*) do not check whether $M > \mathrm{SUM}(A_n)$. In this situation, *Knapsack*(2*a*), *Knapsack*(2*b*) and *Knapsack*(2*c*) are far superior to *Knapsack*(1*a*) and *Knapsack*(1*b*).

The third type of data set contains also random numbers except that they lie in the range [1 .. 1000]. Let's observe Table 6 for the item, $M = \max$ and $n = 20$. *Knapsack*(2)'s need surprisingly less memory because of $Q = 1$ in this item. By the way, *Knapsack*(2)'s are more sensitive than *Knapsack*(1)'s to the value M in computation time because the execution time of the bit-shifting

Table 4. Random number (1 – 100) (times in milliseconds)

M	n	1a	1b	1c	2a	2b	2c
A	20	83.3	< 16.7	16.7	16.7	16.7	16.7
	35	383.3	50.0	100.0	100.0	66.7	33.3
	40	800.0	66.7	150.0	250.0	116.7	50.0
	60	6633.1	500.0	550.0	2083.2	500.0	233.3
	99	—	6299.7	5816.4	39,048.4	2299.9	800.0
B	20	3116.5	83.3	500.0	766.6	400.0	100.0
	25	—	950.0	2816.6	18,815.9	1250.0	366.7
	30		20,015.9	—	—	3616.5	1050.0
	35		—			15,099.7	2449.9
	40					15,099.4	4366.5
	70					†	42,248.3
C	20	13,182.8	116.7	916.6	2283.2	566.6	133.3
	25	—	2083.2	6566.4	67,514.0	2233.2	450.0
	30		—	—	—	5916.4	1183.3
	35					12,999.5	2649.9
	40					24,565.7	4449.8
	70					†	37,848.5
D	20	24,182.4	83.3	1316.6	766.6	300.0	100.0
	25	—	1016.6	6549.7	18,815.9	1250.0	366.7
	30		20,632.5	—	—	3616.5	1050.0
	35		—			7899.7	2449.9
	40					15,099.4	4366.5
	70					†	42,248.3
E	20	28,082.2	50.0	1899.9	< 16.7	< 16.7	< 16.7
	25	—	266.7	8216.3	< 16.7	< 16.7	< 16.7
	30		1466.6	68.413.9	< 16.7	< 16.7	< 16.7
	35		8949.6	260,306.3	16.7	16.7	< 16.7
	40		—	†	16.7	16.7	16.7
	99				116.7	100.0	83.3

†Stop due to exceeding 20 min.
—, Stop due to > 20 MByte.
A, $M = \mathrm{MAX}(A_n)$.
B, $M = \mathrm{SUM}(A_n)/3$.
C, $M = \mathrm{SUM}(A_n)/2$.
D, $M = 2\mathrm{SUM}(A_n)/3$.
E, $M = \mathrm{SUM}(A_n) + 1$.

Table 5. Random number (1 – 100) (space in KBytes)

M	n	1a	1b	1c	2a	2b	2c
A	20	28.2	4.2	2.4	3.9	3.4	3.3
	35	130.3	25.6	12.2	25.0	11.9	10.1
	40	313.0	47.4	28.0	63.2	21.5	18.1
	60	2305.8	253.0	223.9	541.7	75.5	46.0
	99	—	3238.4	3922.6	9633.3	266.3	138.0
B	20	1937.5	53.6	45.7	140.4	33.4	24.7
	25	—	642.2	834.4	3191.3	122.0	77.4
	30		—	—	—	307.9	193.7
	35					585.0	409.4
	40					974.4	678.7
	70					†	4788.4
C	20	8220.3	74.2	72.6	420.8	47.5	33.7
	25	—	1400.7	2022.9	9195.7	154.8	96.4
	30		—	—	—	332.2	224.4
	35					603.9	450.9
	40					981.4	714.0
	70					†	4592.9
D	20	14.475.7	45.5	24.4	140.4	33.4	24.7
	25	—	694.9	843.2	3191.3	122.0	77.3
	30		—	—	—	307.9	193.6
	35					585.0	409.4
	40					974.4	678.7
	70					†	4788.4
E	20	16,384.0	32.0	2.0	0.1	0.1	2.0
	25	—	192.0	5.0	0.1	0.1	3.2
	30		1024.0	12.0	0.2	0.2	4.8
	35		6144.0	32.0	0.2	0.2	6.7
	40		—	†	0.2	0.2	8.6
	99				0.6	0.6	57.1

†Stop due to exceeding 20 min.
—, Stop due to >20 MBytes.
A, $M = \mathrm{MAX}(A_n)$.
B, $M = \mathrm{SUM}(A_n)/3$.
C, $M = \mathrm{SUM}(A_n)/2$.
D, $M = 28\mathrm{SUM}(A_n)/3$.
E, $M = \mathrm{SUM}(A_n)+1$.

procedure is linearly proportional to M. In fact, the goal of the four-table algorithm, *Knapsack*(1*c*), is proposed to find "any one" solution. However, our problem is to find "all" the solutions. It is the reason why the four-table algorithm performs inefficiently.

As a whole, *Knapsack*(2*c*) is the most superior in both time and space. *Knapsack*(2*c*) uses many important properties including symmetric property, data reordering and table-look-up method.

5. FURTHER DISCUSSIONS ON OUR ALGORITHM

In *Knapsack*(1*a*), the set W contains the information of not only $\langle A_n, M\rangle$ but also $\langle A_n, m\rangle$ for $m = 0, 1, \ldots, M$. Obviously, it stores too many informations for a specific $\langle A_n, M\rangle$. In the case of $M = \mathrm{SUM}(A_n)/2$, *Knapsack*(1*a*) needs $O(2^{n-1})$ space no matter how many solution sets $\langle A_n, M\rangle$ has. It is an interesting problem that how many space is needed, using a DAG, to store the solution the solution sets of $\langle A_n, m\rangle$, $m = 0, 1, \ldots, M$. We will claim that we can save the information as much as *Knapsack*(1*a*) with the space bounded by $O(M\cdot n)$, based on the pigeonhole principle. Thus, the upper bound, $O(\mathrm{SUM}(A_n)\cdot n/2)$, is needed to save all the 2^n subsets. This bound is a satisfactory result for the case of $M \ll 2^n$,. From the results in Tables 2, 4 and 7, we believe that our algorithm *Knapsack*(2*b*) is superior to *Knapsack*(1*a*) and (1*b*) under the space consideration in many applications of the VIKP.

6. CONCLUSIONS

In this paper, we propose an efficient pruning algorithm for the VIKP. Firstly, we build a directed tree based on the partition approach to describe the fundamental conception of our algorithm. Secondly, we explain there exist many redundancies in space for *Knapsack*(2*a*) by applying pigeonhole principle and propose a method to remove these redundancies completely. The method

Table 6. Random number (1–1000) (space in KBytes)

M	n	1a	1b	1c	2a	2b	2c
A	20	8.7	1.8	0.8	0.1	0.1	2.8
	35	90.2	9.3	2.9	2.7	2.7	7.3
	40	131.8	14.2	4.4	3.8	3.7	8.9
	60	663.5	78.8	14.4	19.3	14.4	21.3
	99	6160.1	358.6	102.1	223.5	89.0	91.6
B	20	1686.2	27.7	4.9	15.4	12.0	19.4
	25	—	191.5	72.5	343.5	132.5	129.6
	30		1858.0	1648.6	8677.5	792.2	582.6
	40		—	†	—	5392.9	2496.0
	50					†	†
C	20	8194.5	35.7	8.3	40.3	24.0	37.1
	25	—	299.8	185.9	1163.0	275.1	238.6
	30		4149.3	5221.4	†	1227.3	868.9
	40		—	†		†	4552.7
	50						†
D	20	14,702.3	33.7	4.8	15.4	12.0	19.4
	25	—	232.4	72.3	343.5	132.5	129.6
	30		2006.2	1649.0	8677.5	792.2	582.6
	40		—	†	—	5396.9	2496.0
	50					†	†
E	20	16,384.0	32.0	2.0	1.1	1.1	24.0
	25	—	192.0	5.0	1.4	1.4	37.3
	30		1024.0	12.0	1.8	1.8	53.9
	35		6144.0	†	2.1	2.1	75.4
	40		—		2.4	2.4	97.0
	70				4.3	4.3	306.8

†Stop due to >20 min.
—, Stop due to >20 MBytes.
A, $M = \mathrm{MAX}(A_n)$.
B, $M = \mathrm{SUM}(A_n)/3$.
C, $M = \mathrm{SUM}(A_n)/2$.
D, $M = 2\mathrm{SUM}(A_n)/3$.
E, $M = \mathrm{SUM}(A_n)+1$.

Table 7. Random number (1–1000) (times in milliseconds)

M	n	1a	1b	1c	2a	2b	2c
A	20	16.7	<16.7	116.7	<16.7	<16.7	<16.7
	35	250.0	16.7	266.7	83.3	83.3	16.7
	40	400.0	16.7	466.6	100.0	116.7	16.7
	60	2333.2	150.0	2099.9	466.6	400.0	66.7
	99	39,315.1	1133.3	6816.4	4183.2	2699.9	450.0
B	20	2816.6	33.3	2933.2	1150.0	1000.0	50.0
	25	—	300.0	15.699.4	27,798.9	15,482.7	566.6
	30		2866.6	116,612.0	693,272.3	103,462.5	3133.2
	40		—	†	—	764,669.4	16,699.3
	50					†	†
C	20	13,982.8	66.7	7149.7	4616.5	3316.5	116.7
	25	—	466.6	37,048.5	133,828.0	48,731.4	1083.3
	30		6399.7	303,604.5	†	251,073.3	4699.8
	40		—	†		†	30,165.5
	50						†
D	20	25,982.3	66.7	11,449.0	1150.0	1000.0	50.0
	25	—	350.0	57,664.6	27,798.9	15,482.7	566.6
	30		3066.5	468,014.6	693,272.3	103.462.5	3133.2
	40		—	†	—	764,669.4	16,699.3
	50					†	†
E	20	28,065.5	66.7	20,165.9	16.7	33.3	33.3
	25	—	300.0	99,712.7	66.7	66.7	50.0
	30		1533.3	807,984.0	83.3	83.3	83.3
	35		9099.6	†	116.7	100.0	100.0
	50		—		233.3	233.3	216.7
	70				466.6	466.6	450.0

†Stop due to >20 min.
—, Stop due to >20 MBytes.
A, $M = \mathrm{MAX}(A_n)$.
B, $M = \mathrm{SUM}(A_n)/3$.
C, $M = \mathrm{SUM}(A_n)/2$.
D, $M = 2\mathrm{SUM}(A_n)/3$.
E, $M = \mathrm{SUM}(A_n)+1$.

will turn the directed tree into a DAG, which is the major improvement of algorithm *Knapsack*(2*b*). Thirdly, we discuss the influence of different data orderings. In *Knapsack*(2*c*), we sort the data set A_n in increasing order and use table-look-up method to reduce execution time.

By the way, we find a symmetric property. This property makes many heuristic algorithms have a chance to reduce the requirements for computation time and space. We compare our algorithm with the algorithms given in Refs [1,2]. From these experimental results, we verify the excellence of our algorithm. Our algorithm is far superior to others in all 3 types of data set both in computation time and space usage. We also prove that it can store all the information of 2^n subsets within the upper bound, $\mathrm{SUM}(A_n)\cdot n/2$, for space, which is far less than 2^{n-1} using *Knapsack*(1*a*) for the case of $M \ll 2^n$.

REFERENCES

1. E. Horowitz and S. Sahni, Computing partitions with applications to the knapsack problem. *J. ACM* **21**, 277–292 (1974).
2. P. J. Kolesar, A branch and bound algorithm for the knapsack problem. *Manage. Sci.* **May**, 723–735 (1967).
3. R. Schroeppel and A. Shamir, A $T = O(2^{n/2})$, $S = O(2^{n/4})$ algorithm for certain NP-complete problems. *SIAM J. Comput.* **10**, 456–464 (1981).
4. E. D. Karnin, A parallel algorithm for the knapsack problem, *IEEE Trans. Comput.* **C-33**, 404–408 (1984).
5. A. G. Ferreira, A parallel time/hardware tradeoff $T\cdot H = O(2^{n/2})$ for knapsack problem. *IEEE Trans. Comput.* **40**, 221–225 (1991).
6. S. Martello and P. Toth, A mixture of dynamic programming and branch-and-bound for the subset-sum problem. *Manage. Sci.* **30**, 765–771 (1984).
7. J. H. Ahrens and G. Finke, Merging and sorting applied to zero-one knapsack problem. *Oper. Res.* **23**, 1099–1109 (1975).
8. B. Faaland, Solution of the value independent knapsack problem. *Oper. Res.* **21**, 332–337 (1973).
9. S. Martello and P. Toth, *Knapsack Problems, Algorithms and Computer Implementations.* Wiley, Chichester (1990).
10. O. H. Ibrarra and C. E. Kim, Fast approximation algorithm for the knapsack and sum of the subset problems. *JACM.* **22**, 463–468 (1975).