4

Subset-sum problem

4.1 INTRODUCTION

The *Subset-Sum Problem* (SSP) is: given a set of n *items* and a *knapsack*, with

$$w_j = \textit{weight} \text{ of item } j;$$

$$c = \textit{capacity} \text{ of the knapsack},$$

select a subset of the items whose total weight is closest to, without exceeding, c, i.e.

$$\text{maximize} \quad z = \sum_{j=1}^{n} w_j x_j \tag{4.1}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \leq c, \tag{4.2}$$

$$x_j = 0 \text{ or } 1, \qquad j \in N = \{1, \ldots, n\}. \tag{4.3}$$

where

$$x_j = \begin{cases} 1 & \text{if item } j \text{ is selected;} \\ 0 & \text{otherwise.} \end{cases}$$

The problem is related to the *diophantine equation*

$$\sum_{j=1}^{n} w_j x_j = \hat{c}, \tag{4.4}$$

$$x_j = 0 \text{ or } 1, \qquad j = 1, \ldots, n, \tag{4.5}$$

in the sense that the optimal solution value of SSP is the largest $\hat{c} \leq c$ for which (4.4)–(4.5) has a solution.

SSP, which is also called the *Value Independent Knapsack Problem* or *Stickstacking Problem,* is a particular case of the 0-1 knapsack problem (Chapter 2)—arising when $p_j = w_j$ for all j—hence, without loss of generality, we will assume that

$$w_j \text{ and } c \text{ are positive integers,} \tag{4.6}$$

$$\sum_{j=1}^{n} w_j > c, \tag{4.7}$$

$$w_j < c \text{ for } j \in N. \tag{4.8}$$

Violation of such assumptions can be handled as indicated in Section 2.1.

The problem arises in situations where a quantitative target should be reached, such that its negative deviation (or loss of, e.g., trim, space, time, money) must be minimized and a positive deviation is not allowed. Recently, massive SSP's have been used in several coefficient reduction procedures for strengthening LP bounds in general integer programming (see Dietrich and Escudero (1989a, 1989b)).

SSP can obviously be solved (either exactly or heuristically) by any of the methods described in Chapter 2 for the 0-1 knapsack problem. It deserves, however, specific treatment since specialized algorithms usually give much better results. A macroscopic reason for this is the fact that all upper bounds of Sections 2.2 and 2.3 give, for SSP, the trivial value c (since $p_j/w_j = 1$ for all j). SSP can be seen, in fact, as the extreme case of correlation between profits and weights (see Section 2.10). As a consequence, one would even expect catastrophic behaviour of the branch-and-bound algorithms for the 0-1 knapsack problem, degenerating, for SSP, into complete enumeration (because of the value c produced, at all decision nodes, by upper bound computations). This is not always true. In fact, as soon as a feasible solution of value c is determined, one can obviously stop execution and, as we will see, this phenomenon often occurs for problems in which the number of items is not too small. Also note that the reduction procedures of Section 2.7 have no effect on SSP, because of the bound's uselessness.

We describe exact and approximate algorithms for SSP in Sections 4.2 and 4.3, respectively, and analyse computational results in Section 4.4.

4.2 EXACT ALGORITHMS

4.2.1 Dynamic programming

Given a pair of integers m ($1 \leq m \leq n$) and $\hat{c}$ ($0 \leq \hat{c} \leq c$), let $f_m(\hat{c})$ be the optimal solution value of the sub-instance of SSP consisting of items $1, \ldots, m$ and capacity $\hat{c}$. The dynamic programming recursion for computing $f_n(c)$ (optimal solution value of SSP) can be easily derived from that given in Section 2.6 for the 0-1 knapsack problem:

$$f_1(\hat{c}) = \begin{cases} 0 & \text{for } \hat{c} = 0, \ldots, w_1 - 1; \\ w_1 & \text{for } \hat{c} = w_1, \ldots, c; \end{cases}$$

for $m = 2, \ldots, n$:

$$f_m(\hat{c}) = \begin{cases} f_{m-1}(\hat{c}) & \text{for } \hat{c} = 0, \ldots, w_m - 1; \\ \max(f_{m-1}(\hat{c}),\ f_{m-1}(\hat{c} - w_m) + w_m) & \text{for } \hat{c} = w_m, \ldots, c. \end{cases}$$

The time and space complexity to compute $f_n(c)$ is thus $O(nc)$.

Faaland (1973) has presented a specialized dynamic programming approach of the same complexity, which is also suitable for the *bounded* version of SSP, defined by (4.1), (4.2) and

$$0 \leq x_j \leq b_j, \qquad j = 1, \ldots, n,$$

$$x_j \text{ integer}, \qquad j = 1, \ldots, n.$$

The algorithm derives from a recursive technique given by Verebriusova (1904) to determine the number of non-negative integer solutions to diophantine equations (4.4).

Ahrens and Finke (1975) proposed a more effective approach which reduces, on average, the time and space required to solve the problem. The method derives from their dynamic programming algorithm for the 0-1 knapsack problem (Section 2.6.2) and makes use of the "replacement selection" technique, described in Knuth (1973), in order to combine the partial lists obtained by partitioning the variables into four subsets.

Because of the large core memory requirements (the Ahrens and Finke (1975) algorithm needs about $2^{n/4+4}$ words) dynamic programming can be used only for small instances of the problem.

Martello and Toth (1984a) used "partial" dynamic programming lists to obtain a hybrid algorithm (described in the next section) to effectively solve also large instances of SSP. These lists are obtained through a recursion conceptually close to procedure REC2 given in Section 2.6.1 for the 0-1 knapsack problem, but considering only states of total weight not greater than a given value $\tilde{c} \leq c$. The particular structure of SSP produces considerable simplifications. The undominated states are in this case those corresponding to values of $\hat{c}$ for which the diophantine equation (4.4)–(4.5) has a solution. At stage m, the undominated states are determined from the following information, relative to the previous stage:

$$s = \text{number of states at the previous stage;} \tag{4.9}$$

$$b = 2^{m-1}; \tag{4.10}$$

$$W1_i = \text{ total weight of the } i\text{th state } (i = 1, \ldots, s); \tag{4.11}$$

$$X1_i = \{x_1, x_2, \ldots, x_{m-1}\} \qquad \text{for } i = 1, \ldots, s, \tag{4.12}$$

where x_j defines the value of the jth variable in the solution relative to the ith state, i.e. $W1_i = \sum_{j=1}^{m-1} w_j x_j$. Vector $W1_i$ is assumed to be ordered according to strictly increasing values. The procedure updates values (4.9) and (4.10), and stores the new values of (4.11) and (4.12) in $(W2_k)$ and $(X2_k)$. Sets $X1_i$ and $X2_k$ are encoded as bit strings. Note that, for SSP, states having the same weight are equivalent, i.e. dominating each other. In such situations, the algorithm stores only one state, so vector $(W2_k)$ results are ordered according to strictly increasing values. On input, it is assumed that $W1_0 = X1_0 = 0$.

```
procedure RECS:
input: s, b, (W1_i), (X1_i), w_m, c̃;
output: s, b, (W2_k), (X2_k);
begin
    i := 0;
    k := 0;
    h := 1;
    y := w_m;
    W1_{s+1} := +∞;
    W2_0 := 0;
    X2_0 := 0;
    while min(y, W1_h) ≤ c̃ do
        begin
            k := k + 1;
            if W1_h ≤ y then
                begin
                    W2_k := W1_h;
                    X2_k := X1_h;
                    h := h + 1
                end
            else
                begin
                    W2_k := y;
                    X2_k := X1_i + b
                end
            if W2_k = y then
                begin
                    i := i + 1;
                    y := W1_i + w_m
                end
        end
    s := k;
    b := 2b
end.
```

Procedure RECS is a part of the hybrid algorithm described in the next section. It can also be used, however, to directly solve SSP as follows.

```
procedure DPS:
input: n . c . (w_j);
output: z . (x_j);
begin
    c̃ := c;
    W 1_0 := 0;
    X 1_0 := 0;
    s := 1;
    b := 2;
    W 1_1 := w_1;
    X 1_1 := 1;
    m := 2;
    repeat
        call RECS;
        rename W 2 and X 2 as W 1 and X 1, respectively;
        m := m + 1
    until m > n or W 1_s = c;
    z := W 1_s;
    determine (x_j) by decoding X 1_s
end.
```

The time complexity of RECS is $O(s)$. Since s is bounded by min $(2^m - 1 . \tilde{c})$, the time complexity of DPS is $O(\min(2^{n+1} . nc))$.

4.2.2 A hybrid algorithm

Martello and Toth (1984a) used a combination of dynamic programming and tree-search to effectively solve SSP. Assume that the items are sorted beforehand so that

$$w_1 \geq w_2 \geq \cdots \geq w_n . \tag{4.13}$$

The algorithm starts by applying the dynamic programming recursion to a subset containing the last (small) items and by storing the corresponding state lists. Tree-search is then performed on the remaining (large) items. In this way, the state weights in the lists are small and close to each other, while, in the branch-decision tree, the current residual capacity $\hat{c}$ takes small values after few forward moves, allowing use of the dynamic programming lists.

The algorithm starts by determining two partial state lists:

(i) given a prefixed value $MA < n - 1$, list $(WA_i . XA_i)$. $i = 1, \ldots, SA$, contains all the undominated states induced by the last MA items;

(ii) given two prefixed values MB $(MA < MB < n)$ and $\overline{c}$ $(w_n < \overline{c} < c)$, list $(WB_i . XB_i)$. $i = 1, \ldots, SB$, contains the undominated states of weight not greater than $\overline{c}$ induced by the last MB items.

Figure 4.1, in which $NA = n - MA + 1$ and $NB = n - MB + 1$, shows the states covered by the two lists: the thick lines approximate the step functions giving, for each item, the maximum state weight obtained at the corresponding iteration.

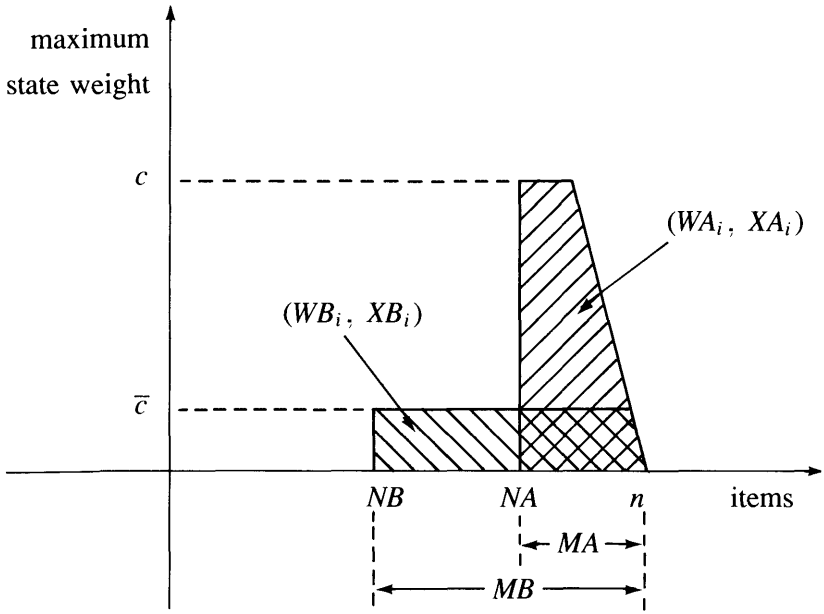

Figure 4.1 States covered by the dynamic programming lists

The following procedure determines the two lists. List (WA_i, XA_i) is first determined by calling procedure RECS in reverse order, i.e. determining, for $m = n, n-1, \ldots, NA(\equiv n - MA + 1)$, the optimal solution value $\varphi_m(\hat{c})$ of the sub-instance defined by items $m, m+1, \ldots, n$ and capacity $\hat{c} \leq c$. List (WB_i, XB_i) is then initialized to contain those states of (WA_i, XA_i) whose weight is not greater than $\bar{c}$, and completed by calling RECS for $m = NA - 1, NA - 2, \ldots, NB(\equiv n - MB + 1)$. Note that the meaning of XA and XB is consequently altered with respect to (4.12).

procedure LISTS:
input: $n, c, (w_j), NA, NB, \bar{c}$;
output: $SA, (WA_i), (XA_i), SB, (WB_i), (XB_i)$;
begin
comment: determine list (WA_i, XA_i);
$\tilde{c} := c$;
$W1_0 := 0$;
$X1_0 := 0$;
$s := 1$;
$b := 2$;
$W1_1 := w_n$;
$X1_1 := 1$;

```
    m := n − 1;
    repeat
        call RECS;
        rename W2 and X2 as W1 and X1, respectively;
        m := m − 1
    until m < NA or W1_s = c;
    for i := 1 to s do
        begin
            WA_i := W1_i;
            XA_i := X1_i
        end;
    SA := s;
    if WA_SA < c then (comment: determine list (WB_i . XB_i))
        begin
            c̃ := c̄;
            determine, through binary search, ī = max{i : WA_i ≤ c̄};
            s := ī;
            repeat
                call RECS;
                rename W2 and X2 as W1 and X1, respectively;
                m := m − 1
            until m < NB;
            rename W1 and X1 as WB and XB, respectively;
            SB := s
        end
end.
```

Example 4.1

Consider the instance of SSP defined by

$n = 10;$

$(w_j) = (41, 34, 21, 20, 8, 7, 7, 4, 3, 3);$

$c = 50;$

$\text{MA} = 4;$

$\text{MB} = 6;$

$\bar{c} = 12.$

Calling LISTS, we obtain $SA = 9$. $SB = 8$ and the values given in Figure 4.2. □

We can now state the overall algorithm. After having determined the dynamic programming lists, the algorithm generates a binary decision-tree by setting $\hat{x}_j$ to 1 or 0 for $j = 1, \ldots, NA - 1$. Only the first $NA - 1$ items are considered, since all the feasible combinations of items $NA, \ldots, n$ are in list $(WA_i . XA_i)$. A forward

i	WA_i	XA_i(decoded)	WB_i	XB_i(decoded)
0	0	0	0	0
1	3	1	3	1
2	4	100	4	100
3	6	11	6	11
4	7	101	7	101
5	10	111	8	100000
6	11	1100	10	111
7	13	1011	11	1100
8	14	1101	12	100100
9	17	1111		

Figure 4.2 Dynamic programming lists for Example 4.1

move starting from an item j consists in: (a) finding the first item $j' \geq j$ which can be added to the current solution; (b) adding to the current solution a feasible sequence $j'.j'+1, \ldots, j''$ of consecutive items until the residual capacity $\hat{c}$ is no greater than $\overline{c}$. A backtracking step consists in removing from the current solution that item j''' which was inserted last and in performing a forward move starting from $j'''+1$.

At the end of a forward move, we determine the maximum weight δ of a dynamic programming state which can be added to the current solution. This is done by assuming the existence of two functions, A and B, to determine, respectively,

$$A(\hat{c}) = \max_{0 \leq i \leq SA}\{i : WA_i \leq \hat{c}\},$$

$$B(\hat{c},j) = \max_{0 \leq i \leq SB}\{i : WB_i \leq \hat{c} \text{ and } y_k^i = 0 \text{ for all } k < j\},$$

where (y_k^i) denotes the binary vector encoded in XB_i. (Both $A(\hat{c})$ and $B(\hat{c}.j)$ can be implemented through binary search.) After updating of the current optimal solution z ($z := \max(z,(c-\hat{c})+\delta)$), we proceed to the next forward move, unless we find that the solution values of all the descendent decision nodes are dominated by $(c-\hat{c})+\delta$. This happens when either the next item which we could insert is one of the MA last items, or is one of the MB last items and the residual capacity $\hat{c}$ is no greater than $\overline{c}$.

Values $F_k = \sum_{j=k}^{n} w_j$ $(k = 1, \ldots, n)$ are used to avoid forward moves when $\hat{c} \geq F_{j'}$ or an upper bound on the optimal solution obtainable from the move is no greater than the value of the best solution so far.

procedure MTS;
input: $n, c, (w_j), \bar{c}, MA, MB$;
output: $z, (x_j)$;
begin
1. [initialize]
$NA := n - MA + 1$;
$NB := n - MB + 1$;
call LISTS;
$z := WA_{SA}$;
for $k := 1$ **to** $NA - 1$ **do** $x_k := 0$;
let (y_k) be the binary vector encoded in XA_{SA};
for $k := NA$ **to** n **do** $x_k := y_k$;
if $z = c$ **then return**;
for $k := n$ **to** 1 **step** -1 **do** compute $F_k = \sum_{j=k}^{n} w_j$;
$\hat{z} := 0$;
$\hat{c} := c$;
for $k := 1$ **to** n **do** $\hat{x}_k := 0$;
$j := 1$;
2. [try to avoid the next forward move]
while $w_j > \hat{c}$ and $j < NA$ **do** $j := j + 1$;
if $j = NA$ **then go to** 4;
if $F_j \leq \hat{c}$ **then**
begin
if $\hat{z} + F_j > z$ **then** (**comment**: new optimal solution)
begin
$z := \hat{z} + F_j$;
for $k := 1$ **to** $j - 1$ **do** $x_k := \hat{x}_k$;
for $k := j$ **to** n **do** $x_k := 1$;
if $z = c$ **then return**
end;
go to 5
end;
determine, through binary search, $r = \min\{k > j : F_k \leq \hat{c}\}$;
$s := n - r + 1$;
comment: at most s items can be added to the current solution;
$u := F_j - F_{j+s}$;
comment: $u = \sum_{k=j}^{j+s-1} w_j$ = total weight of the s largest available items;
if $\hat{z} + u \leq z$ **then go to** 5;
3. [perform a forward move]
while $w_j \leq \hat{c}$ and $j < NA$ and $\hat{c} > \bar{c}$ **do**
begin
$\hat{c} := \hat{c} - w_j$;
$\hat{z} := \hat{z} + w_j$;
$\hat{x}_j := 1$;
$j := j + 1$
end;

```
4. [use the dynamic programming lists]
    if ĉ ≤ c̄ then
        begin
            δ := WB_{B(ĉ,j)};
            flag := "b"
        end
    else
        begin
            δ := WA_{A(ĉ)};
            flag := "a";
            if δ < c̄ and z < ẑ + c̄ then
                begin
                    δ := WB_{B(ĉ,j)};
                    flag := "b"
                end
        end;
    comment: δ is the maximum additional weight obtainable from the lists;
    if ẑ + δ > z then (comment: update the optimal solution)
        begin
            z := ẑ + δ;
            for k := 1 to j − 1 do x_k := x̂_k;
            if flag = "a" then
                begin
                    for k := j to NA − 1 do x_k := 0;
                    let (y_k) be the vector encoded in XA_{A(ĉ)};
                    for k := NA to n do x_k := y_k
                end
            else
                begin
                    for k := j to NB − 1 do x_k := 0;
                    let (y_k) be the vector encoded in XB_{B(ĉ,j)};
                    for k := max(NB, j) to n do x_k := y_k
                end;
            if z = c then return
        end;
    if (ĉ < w_{NA−1} or j = NA) then go to 5;
    if (ĉ < w_{NB−1} or j ≥ NB) and (ĉ < c̄) then go to 5
    else go to 2;
5. [backtrack]
    find i = max{k < j : x̂_k = 1};
    if no such i then return;
    ĉ := ĉ + w_i;
    ẑ := ẑ − w_i;
    x̂_i := 0;
    j := i + 1;
    go to 2
end.
```

Example 4.1 (continued)

Executing MTS, we obtain:

NA = 7,

NB = 5,

(F_k) = (148, 107, 73, 52, 32, 24, 17, 10, 6, 3),

the dynamic programming lists of Figure 4.2 and the branch-decision tree of Figure 4.3. □

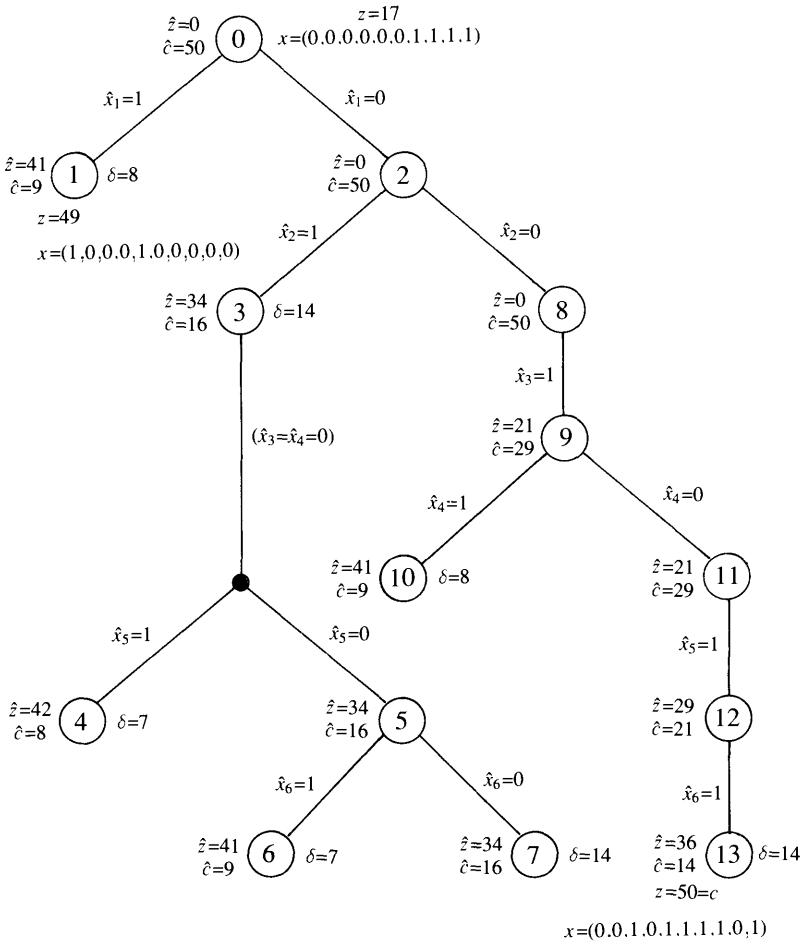

Figure 4.3 Branch-decision tree of Procedure MTS for Example 4.1

The Fortran implementation of procedure MTS is included in that of procedure MTSL, which is described in the next section. The parameters for the dynamic programming lists must take into account the "difficulty" of the problem. They have been experimentally determined as the following functions of n and $wmax = \max\{w_j\}$:

$$MA = \min(2\log_{10} wmax,\ 0.7n);$$

$$MB = \min(2.5\log_{10} wmax,\ 0.8n);$$

$$\bar{c} = 1.3w_{NB}.$$

These values are automatically decreased by the code corresponding to MTS whenever the space required from the lists is larger than the available core memory.

A different hybrid algorithm for SSP can be found in Plateau and Elkihel (1985).

4.2.3 An algorithm for large-size problems

Computational experiments with algorithm MTS show (Martello and Toth, 1984a) that many instances of SSP can be exactly solved in reasonable computing time, since they admit a large number of feasible solutions of value c (i.e. optimal). Hence, for large-size problems, there is the possibility of finding one such solution by considering (and sorting) only a relatively small subset of the items. This can be obtained by defining a *core problem* which has a structure similar to that introduced for the 0-1 knapsack problem (Section 2.9) but can be determined much more efficiently as follows. Given an instance of SSP, we determine the *critical item* $s = \min\{j : \sum_{i=1}^{j} w_i > c\}$ and, for a prefixed value $\vartheta > 0$, we define the *core problem*

$$\text{maximize} \quad \tilde{z} = \sum_{j=s-\vartheta}^{s+\vartheta} w_j x_j \tag{4.14}$$

$$\text{subject to} \quad \sum_{j=s-\vartheta}^{s+\vartheta} w_j x_j \leq \tilde{c} = c - \sum_{j=1}^{s-\vartheta-1} w_j. \tag{4.15}$$

$$x_j = 0 \text{ or } 1. \qquad j = s-\vartheta, \ldots, s+\vartheta. \tag{4.16}$$

Then we sort items $s-\vartheta, \ldots, s+\vartheta$ according to (4.13) and solve the core problem through procedure MTS. If the solution value found is equal to $\tilde{c}$ then we have an optimal solution of value c for SSP, defined by values $x_{s-\vartheta}, \ldots, x_{s+\vartheta}$ returned by MTS, and by $x_j = 1$ for $j < s-\vartheta$, $x_j = 0$ for $j > s+\vartheta$. Otherwise, we enlarge the core problem by increasing ϑ and repeat.

```
procedure MTSL:
input: n, c, (w_j), ϑ, MA, MB, c̄;
output: z, (x_j);
begin
    determine s = min{j : Σ_{i=1}^{j} w_i > c};
    repeat
        a := max(1, s − ϑ);
        b := min(n, s + ϑ);
        c̃ := c − Σ_{j=1}^{a−1} w_j;
        sort items a, a + 1, ..., b according to decreasing weights;
        call MTS for the core problem (4.14)–(4.16) and let z̃ be the solution
          value returned;
        ϑ := 2ϑ
    until z̃ = c̃ or b − a + 1 = n;
    let y_j (j = a, ..., b) be the solution vector returned by MTS;
    for j := 1 to a − 1 do x_j := 1;
    for j := a to b do x_j := y_j;
    for j := b + 1 to n do x_j := 0;
    z := z̃ + (c − c̃)
end.
```

A "good" input value for ϑ was experimentally determined as

$$\vartheta = 45.$$

The Fortran implementation of MTSL is included in the present volume.

4.3 APPROXIMATE ALGORITHMS

4.3.1 Greedy algorithms

The most immediate approach to the heuristic solution of SSP is the *Greedy Algorithm,* which consists in examining the items in any order and inserting each new item into the knapsack if it fits. By defining $p_j = w_j$ for all j, we can use procedure GREEDY given in Section 2.4 for the 0-1 knapsack problem. This procedure will consider, for SSP, the item of maximum weight alone as a possible alternative solution, and guarantee a worst-case performance ratio equal to $\frac{1}{2}$. No sorting being needed, since (2.7) is satisfied by any instance of SSP, the time complexity decreases from $O(n\log n)$ to $O(n)$.

For SSP, better average results can be obtained by sorting the items according to decreasing weights. Since in this way the item of maximum weight is always considered first (and hence inserted), we no longer need to explicitly determine it, so a considerably simpler procedure is the following. We assume that the items are sorted according to (4.13).

```
procedure GS:
input: n, c, (w_j);
output: z^g, (x_j);
begin
    ĉ := c;
    for j := 1 to n do
        if w_j > ĉ then x_j := 0
        else
            begin
                x_j := 1;
                ĉ := ĉ − w_j
            end;
    z^g := c − ĉ
end.
```

The worst-case performance ratio is still $\frac{1}{2}$, while the time complexity grows to $O(n\log n)$ because of the required sorting.

An $O(n^2)$ greedy algorithm, with better worst-case performance ratio was given by Martello and Toth (1984b). The idea is to apply the greedy algorithm n times, by considering item sets $\{1,\ldots,n\}$, $\{2,\ldots,n\}$, $\{3,\ldots,n\}$, and so on, respectively, and take the best solution. Assuming that the items are sorted according to (4.13), the algorithm is the following.

```
procedure MTGS:
input: n, c, (w_j);
output: z^g, X^h;
begin
    z^g := 0;
    for i := 1 to n do
        begin
            ĉ := c;
            Y := Ø;
            for j := i to n do
                if w_j ≤ ĉ then
                    begin
                        ĉ := ĉ − w_j;
                        Y := Y ∪ {j}
                    end;
            if c − ĉ > z^g then
                begin
                    z^g := c − ĉ;
                    X^h := Y;
                    if z^g = c then return
                end
        end
end.
```

The time complexity of MTGS is clearly $O(n^2)$. Its worst-case performance ratio is established by the following

Theorem 4.1 (Martello and Toth, 1984b) $r(\text{MTGS}) = \frac{3}{4}$.

Proof. We will denote by $z(k)$ the value $c - \hat{c}$ of the solution found by the algorithm at the kth iteration, i.e. by considering item set $\{k, \ldots, n\}$. Let

$$q = \max \{j : \exists\, k < j \text{ such that item } j \text{ is not selected for } z(k)\}. \tag{4.17}$$

$$Q = \sum_{j=q+1}^{n} w_j, \tag{4.18}$$

and note that, because of (4.17), items $q+1, \ldots, n$ are selected for all $z(k)$ with $k \leq q+1$. Let $z = \sum_{j=1}^{n} w_j x_j^*$ be the optimal solution value and define $A = \{j \leq q : x_j^* = 1\}$.

(a) If $|A| \leq 2$ then $z^g = z$. In fact: (i) if $|A| = 1$, with $A = \{j_1\}$, we have $z^g \geq z(j_1) \geq w_{j_1} + Q = z$; (ii) if $|A| = 2$, with $A = \{j_1, j_2\}$ and $j_1 < j_2$, we have $z^g \geq z(j_1) \geq w_{j_1} + w_{j_2} + Q = z$.

(b) If $|A| > 2$ then $z^g \geq \frac{3}{4}c \geq \frac{3}{4}z$. In fact: (i) if $w_q > \frac{1}{4}c$, we have $z^g \geq z(q-2) = w_{q-2} + w_{q-1} + w_q + Q > \frac{3}{4}c$; (ii) if $w_q \leq \frac{1}{4}c$, there must exist an iteration $\bar{k} \leq q-1$ in which item q is not selected for $z(\bar{k})$ since $w_q > \hat{c} \geq c - z(\bar{k})$, and hence we have $z^g \geq z(\bar{k}) > c - w_q \geq \frac{3}{4}c$.

To prove that value $\frac{3}{4}$ is tight, consider the series of instances with $n = 4$, $w_1 = 2R$, $w_2 = R+1$, $w_3 = w_4 = R$ and $c = 4R$. The optimal solution value is $z = 4R$, while $z(1) = z(2) = 3R+1$, $z(3) = 2R$ and $z(4) = R$, so $z^g = 3R+1$. Hence the ratio z^g/z can be arbitrarily close to $\frac{3}{4}$, for R sufficiently large. □

Note that, for the series of instances above, the optimal solution would have been produced by a modified version of the algorithm applying the greedy search at iteration k to item set $\{k, \ldots, n, 1, \ldots, k-1\}$ (the result would be $z^g = z(3) = 4R$). However, adding a fifth element with $w_5 = 1$ gives rise to a series of problems whereby z^g/z tends to $\frac{3}{4}$ for the modified algorithm as well. Also, from the practical point of view, computational experiments with the modified algorithm (Martello and Toth, 1985a) show very marginal average improvements with considerably higher computing times.

More accurate approximate solutions can obviously be obtained by using any of the approximation schemes described for the 0-1 knapsack problem (Section 2.8). However, by exploiting the special structure of the problem, we can obtain better schemes—both from the theoretical and the practical point of view—for the approximate solution of SSP.

4.3.2 Polynomial-time approximation schemes

The first polynomial-time approximation scheme was given by Johnson (1974). The idea is to identify a subset of "large" items (according to a given parameter k) and to find the corresponding optimal solution. This is completed by applying the greedy algorithm, for the residual capacity, to the remaining items. The algorithm can be efficiently implemented as the following procedure (slightly different from the original one presented by Johnson (1974)), in which k is supposed to be a positive integer:

procedure J(k):
input: $n, c, (w_j)$;
output: z^h, X^h;
begin
  $L := \{j : w_j > c/(k+1)\}$;
  determine $X^h \subseteq L$ such that $z^h = \sum_{j \in X^h} w_j$ is closest to, without exceeding, c;
  $\hat{c} := c - z^h$;
  $S := \{1, \ldots, n\} \backslash L$;
  sort the items in S according to decreasing weights and let $m = \min_{j \in S}\{w_j\}$;
  while $S \neq \emptyset$ and $\hat{c} \geq m$ **do**
    begin
      let j be the first item in S;
      $S := S \backslash \{j\}$;
      if $w_j \leq \hat{c}$ **then**
        begin
          $\hat{c} := \hat{c} - w_j$;
          $X^h := X^h \cup \{j\}$
        end
    end;
    $z^h := c - \hat{c}$
end.

The time complexity to determine the initial value of z^h and the corresponding X^h through complete enumeration is $O(n^k)$, since $|X^h| \leq k$. The remaining part of the algorithm requires time $O(n \log n)$—for sorting—plus $O(n)$. The overall time complexity of J(k) is thus $O(n \log n)$ for $k = 1$, and $O(n^k)$ for $k > 1$. The space complexity is $O(n)$.

Theorem 4.2 (Johnson, 1974) $r(\mathrm{J}(k)) = k/(k+1)$.

Proof. Let $z = \sum_{j \in X^*} w_j$ be the optimal solution value and consider partition of the optimal item set X^* into $L^* = \{j \in X^* : w_j > c/(k+1)\}$ and $S^* = X^* \backslash L^*$. Similarly, partition item set X^h returned by J(k) into $L^h = \{j \in X^h : w_j > c/(k+1)\}$ and $S^h = X^h \backslash L^h$. Since L^h is the optimal subset of $L = \{j : w_j > c/(k+1)\}$,

initially determined by the algorithm, we have $\sum_{j \in L^h} w_j \geq \sum_{j \in L^*} w_j$. Hence, if $S^* \subseteq S^h$, we also have $\sum_{j \in S^h} w_j \geq \sum_{j \in S^*} w_j$ and the solution found by the algorithm is optimal. Otherwise, let $q \in S^*$ be any item not selected for S^h: it must be $w_q + z^h > c$, so $z^h > c - w_q \geq ck/(k+1) \geq zk/(k+1)$.

Tightness of the $k/(k+1)$ bound is proved by the series of problems with $n = k+2$. $w_1 = R+1$. $w_j = R$ for $j > 1$ and $c = (k+1)$. The optimal solution value is $z = (k+1)R$. Since it results that $L = \{1\}$, the heuristic solution is $z^h = kR + 1$, so the ratio z^h/z can be arbitrarily close to $k/(k+1)$ for R sufficiently large. □

Note that J(1) produces the greedy solution. In fact $L = \{j : w_j > c/2\}$, so only one item (the one with largest weight) will be selected from L while, for the remaining items, a greedy search is performed.

Example 4.2

Consider the instance of SSP defined by

$n = 9;$

$(w_j) = (81, 80, 43, 40, 30, 26, 12, 11, 9);$

$c = 100.$

MTGS gives, in $O(n^2)$ time: $z^h = \max(93, 92, 95, 96, 88, 58, 32, 20, 9) = 96$, $X^h = \{4, 5, 6\}$.

J(1) (as well as GS) gives, in $O(n \log n)$ time: $L = \{1, 2\}$, $z^h = 93$, $X^h = \{1, 7\}$.

J(2) gives, in $O(n^2)$ time: $L = \{1, 2, 3, 4\}$, $z^h = 95$, $X^h = \{3, 4, 7\}$.

J(3) gives, in $O(n^3)$ time: $L = \{1, 2, 3, 4, 5, 6\}$, $z^h = 99$, $X^h = \{3, 5, 6\}$.

The optimal solution $z = 100$. $X = \{2, 8, 9\}$ is found by J(11). □

A better polynomial-time approximation scheme has been found by Martello and Toth (1984b) by combining the idea in their algorithm MTGS of the previous section with that in the Sahni (1975) scheme for the 0-1 knapsack problem (see Section 2.8.1). For $k = 2$, the resulting scheme applies MTGS to the original problem (for $k = 1$ the scheme is not defined but it is assumed to be the greedy algorithm). For $k = 3$, it imposes each item in turn and applies MTGS to the resulting subproblem, taking the best solution. For $k = 4$, all possible item pairs are imposed, and so on. It will be shown in Section 4.4.2 that, for practical purposes, $k = 2$ or 3 is enough for obtaining solutions very close to the optimum. It is assumed that the items are sorted according to (4.13).

procedure MTSS(k):
input: $n, c, (w_j)$;
output: z^h, X^h;
begin
$z^h := 0;$

```
for each M ⊂ N = {1,...,n} such that |M| ≤ k − 2 do
    begin
        g := Σ_{j∈M} w_j;
        if g ≤ c then
            begin
                call MTGS for the subproblem defined by item set
                  N\M and reduced capacity c − g, and let z^g =
                  Σ_{j∈V} w_j (V ⊆ N\M) be the solution found;
                if z^g > z^h then
                    begin
                        z^h := z^g;
                        X^h := M ∪ V;
                        if z^h = c then return
                    end
            end
    end
end.
```

Since there are $O(n^{k-2})$ subsets $M \subset N$ of cardinality not greater than $k-2$, and recalling that MTGS requires $O(n^2)$ time, the overall time complexity of MTSS(k) is $O(n^k)$. The space complexity is clearly $O(n)$. From Theorem 4.1 we have $r(\text{MTSS}(2)) = \frac{3}{4}$. Martello and Toth (1984b) have proved that $r(\text{MTSS}(3)) = \frac{6}{7}$ and $(k+3)/(k+4) \leq r(\text{MTSS}(k)) \leq k(k+1)/(k(k+1)+2)$ for $k \geq 4$. Fischetti (1986) exactly determined the worst-case performance ratio of the scheme:

Theorem 4.3 (Fischetti, 1986) $r(\text{MTSS}(k)) = (3k-3)/(3k-2)$.

Proof. We omit the part proving that $r(\text{MTSS}(k)) \geq (3k-3)/(3k-2)$. Tightness of the bound is proved by the series of problems with $n = 2k$, $w_j = 2R$ for $j < k$, $w_k = R+1$, $w_j = R$ for $j > k$ and $c = (3k-2)R$ (e.g., for $k = 4$, $(w_j) = (2R.2R.2R.R+1.R.R.R.R)$. $c = 10R$). The unique optimal solution, of value $z = (3k-2)R$, includes all the items but the kth. Performing MTSS(k), there is no iteration in which M contains all items $j < k$, so the optimal solution could be found only by a greedy search starting from an item $j < k$. All such searches, however, will certainly include item k (since, at each iteration, at least two items of weight R are not in M), hence producing a solution value not greater than the greedy solution value $z^h = z^g = (3k-3)R+1$. It follows that the ratio z^h/z can be arbitrarily close to $(3k-3)/(3k-2)$ for R sufficiently large. □

MTSS(k) dominates the Johnson (1974) scheme J(k), in the sense that, for any $k > 1$, the time complexity of both schemes is $O(n^k)$, while $r(\text{MTSS}(k)) = (3k-3)/(3k-2) > k/(k+1) = r(\text{J}(k))$ (for example: $r(\text{MTSS}(2)) = \frac{3}{4} = r(\text{J}(3))$, $r(\text{MTSS}(3)) = \frac{6}{7} = r(\text{J}(6))$, $r(\text{MTSS}(4)) = \frac{9}{10} = r(\text{J}(9))$). Also note that, for increasing values of k, the solution values returned by MTSS(k) are non-decreasing (because of the definition of M), while those returned by J(k) are not (if, for

example, $(w_j) = (8.5.5.3)$ and $c = 12$, J(1) returns $z^h = 11$, while J(2) returns $z^h = 10$).

Example 4.2 (continued)

We have already seen that MTSS(2) gives, in $O(n^2)$ time: $z^h = 96$. $X^h = \{4.5.6\}$. MTSS(3) gives, in $O(n^3)$ time: $z^h = 100$. $X^h = \{2.8.9\}$ (optimal). The solution is found when $M = \{2\}$ and the greedy search is performed starting from item 8. □

A more effective implementation of MTSS(k) can be obtained if, at each iteration i in the execution of MTGS, we update a pair $(L.\tilde{c})$ having the property that all items in $B = \{i.....n\}\backslash(M \cup L)$ will be selected by the greedy search starting from i, and $\tilde{c} = c - \sum_{j \in B} w_j$. In this way, the greedy search can be performed only for the items in L with residual capacity $\tilde{c}$. Since each iteration removes items from L, execution of MTGS can be halted as soon as $L = \emptyset$. The improved version of MTSS(k) is obtained by replacing the call to MTGS with the statement

call MTGSM,

where:

```
procedure MTGSM:
input: n.c.(w_j).M.g.z^h;
output: z^g.V;
begin
    z^g := z^h;
    L := {1.....n}\M;
    c̃ := c − g;
    S := ∅;
    i := 0;
    repeat
        i := i + 1;
        if i ∉ M then
            begin
                while L ≠ ∅ and w_j ≤ c̃ (j the first item in L) do
                    begin
                        c̃ := c̃ − w_j;
                        S := S ∪ {j};
                        L := L\{j}
                    end;
                ĉ := c̃;
                T := S;
                for each j ∈ L do if w_j ≤ ĉ then
                    begin
                        ĉ := ĉ − w_j;
                        T := T ∪ {j}
                    end;
```

```
                if c − ĉ > z^g then
                    begin
                        z^g := c − ĉ;
                        V := T
                    end;
                c̃ := c̃ + w_i;
                S := S \ {i}
            end;
    until L = Ø or z^g = c
end.
```

Example 4.2 (continued)

Calling MTGSM with $z^h = 0$. $M = \emptyset$ and $g = 0$, the execution is:

$L = \{1, \ldots, 9\}$. $\tilde{c} = 100$, $S = \emptyset$;

$i = 1$: $\tilde{c} = 19$, $S = \{1\}$, $L = \{2, \ldots, 9\}$;
$\hat{c} = 7$, $T = \{1, 7\}$, $z^g = 93$, $V = \{1, 7\}$;
$\tilde{c} = 100$, $S = \emptyset$;

$i = 2$: $\tilde{c} = 20$, $S = \{2\}$, $L = \{3, \ldots, 9\}$;
$\hat{c} = 8$, $T = \{2, 7\}$;
$\tilde{c} = 100$, $S = \emptyset$;

$i = 3$: $\tilde{c} = 17$, $S = \{3, 4\}$, $L = \{5, \ldots, 9\}$;
$\hat{c} = 5$, $T = \{3, 4, 7\}$, $z^g = 95$, $V = \{3, 4, 7\}$;
$\tilde{c} = 60$, $S = \{4\}$;

$i = 4$: $\tilde{c} = 4$, $S = \{4, 5, 6\}$, $L = \{7, 8, 9\}$;
$\hat{c} = 4$, $T = \{4, 5, 6\}$, $z^g = 96$, $V = \{4, 5, 6\}$;
$\tilde{c} = 44$, $S = \{5, 6\}$;

$i = 5$: $\tilde{c} = 12$, $S = \{5, 6, 7, 8, 9\}$, $L = \emptyset$;
$\hat{c} = 12$, $T = \{5, 6, 7, 8, 9\}$;
$\tilde{c} = 42$, $S = \{6, 7, 8, 9\}$. □

For large values of n, the computing time required by MTSS(k) can be further reduced in much the same way used for MTSL (Section 4.2.3), i.e. by determining the solution for an approximate core problem and then checking whether the requested performance (evaluated with respect to upper bound c on z) has been obtained.

Fischetti (1989) has proposed a polynomial-time approximation scheme, FS(k), based on the subdivision of N into a set of "small" items and a number of sets of "large" items, each containing items of "almost equal" weight. Although the worst-case performance ratio of the scheme has not been determined, it has been proved

that $r(\mathrm{FS}(k)) \geq ((k+2)^2-4)/(k+2)^2$. With this ratio, the result is $r(\mathrm{MTSS}(k)) > r(\mathrm{FS}(k))$ for $k < 6$, while $r(\mathrm{MTSS}(k)) < r(\mathrm{FS}(k))$ for $k > 6$.

4.3.3 Fully polynomial-time approximation schemes

The algorithms of the previous section allow one to obtain any prefixed worst-case performance ratio r in polynomial time and with linear space. The time complexity, however, is exponential in the inverse of the worst-case relative error $\varepsilon = 1 - r$.

The fully polynomial-time approximation scheme proposed by Ibarra and Kim (1975) for the 0-1 knapsack problem (procedure IK(ε) of Section 2.8.2) also applies to SSP. No sorting being required, the time complexity decreases from $O(n\log n) + O(n/\varepsilon^2)$ to $O(n/\varepsilon^2)$, polynomial in $1/\varepsilon$, while the space complexity remains $O(n + (1/\varepsilon^3))$. Lawler (1979) adapted to SSP his improved version of the Ibarra and Kim (1975) scheme for the 0-1 knapsack problem, obtaining time complexity $O(n + (1/\varepsilon^3))$ and space complexity $O(n + (1/\varepsilon^2))$, or time and space complexity $O(n + (1/\varepsilon^2)\log(1/\varepsilon))$.

All of the above schemes are based on the same idea, i.e. (see Section 2.8.2): (a) partitioning the items, basing on the value of ε, into "large" and "small" ones; (b) solving the problems for the large items only, with scaled weights, through dynamic programming; (c) completing the solution, in a greedy way, with the small items. Gens and Levner (1978, 1980) have proposed a fully polynomial-time approximation scheme based on a different (and simpler) principle. They solve the complete problem through ordinary dynamic programming but, at each iteration, reduce the current dynamic programming lists by keeping only state weights differing from each other by at least a threshold value depending on ε. The scheme can be conveniently defined using procedure RECS of Section 4.2.1. Note that the algorithm results similar to procedure DPS for the exact dynamic programming solution of SSP (Section 4.2.1). The main difference consists in determining, after each RECS call, reduced lists $W1$ and $X1$, instead of simply renaming $W2$ and $X2$ as $W1$ and $X1$.

procedure GL(ε):
input: $n, c, (w_j)$;
output: z^h, X^h;
begin
determine $\sigma = \max\{j : \sum_{i=1}^{j} w_i \leq c\}$;
$\tilde{z} := \max(\sum_{j=1}^{\sigma} w_j, \max_j\{w_j\})$ (**comment**: $\tilde{z} \leq z < 2\tilde{z}$);
$\tilde{c} := c$;
$W1_0 := 0$;
$X1_0 := 0$;
$s := 1$;
$b := 2$;
$W1_1 := w_1$;
$X1_1 := 1$;
$m := 2$;
repeat

```
        call RECS ;
        h := 0;
        j := 0;
        repeat
            if W2_{j+1} > W1_h + εz̃ then j := j + 1
            else j := max{q : W2_q ≤ W1_h + εz̃};
            h := h + 1;
            W1_h := W2_j;
            X1_h := X2_j
        until j = s;
        m := m + 1;
        s := h
    until m > n or W1_s = c;
    z^h := W1_s;
    determine X^h by decoding X1_s
end.
```

At each iteration, the reduced dynamic programming lists clearly satisfy $W1_{h+2} - W1_h > \varepsilon\tilde{z}$ for $h = 1, \ldots, s-2$. Hence the number of states is always bounded by $s < 2z/(\varepsilon\tilde{z})$, that is, from $z < 2\tilde{z}$, by $s < (4/\varepsilon)$. It follows that the scheme has time and space complexity $O(n/\varepsilon)$. The proof that the solution determined by GL(ε) has worst-case relative error not greater than ε is given in Levner and Gens (1978) and Gens and Levner (1978).

The time and space complexity of GL(ε) can be better or worse than that of the Lawler (1979) scheme, according to the values of n and ε.

Example 4.2 (continued)

Calling GL($\frac{1}{3}$) , we initially find $\sigma = 1$. $\tilde{z} = 81$ and the weight list $W1$ given in the first column of Figure 4.4. No state is eliminated for $m = 2.3$. For $m = 4.W2_4$ is eliminated since $W2_5 - W2_3 = 3 \leq \varepsilon\tilde{z} = 27$. The approximate solution found has the final value of $W1_4$, i.e. $z^h = 96$ (with $X^h = \{4.5.6\}$). □

4.3.4 Probabilistic analysis

As for the 0-1 knapsack problem (Section 2.8.3), we give a brief outline of the main results obtained in probabilistic analysis.

The first probabilistic result for SSP was obtained by d'Atri and Puech (1982). Assuming that the weights are independently drawn from a uniform distribution over $\{1, 2, \ldots, c(n)\}$ and the capacity from a uniform distribution over $\{1, 2, \ldots, nc(n)\}$, where $c(n)$ is an upper bound on the weights value, they proved that a simple variant of the greedy algorithm solves SSP with probability tending to 1.

Lagarias and Odlyzko (1983) considered SSP with equality constraint and assumed that the weights are independently drawn from a uniform distribution

	$m = 1$	$m = 2$		$m = 3$		$m = 4$		$m = 5$		$m = 6$		$m = 7$		$m = 8$		$m = 9$	
h	$W1$	$W2$	$W1$	$W2$	$W1$	$W2$	$W1$	$W2$	$W1$	$W2$	$W1$	$W2$	$W1$	$W2$	$W1$	$W2$	$W1$
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	81	80	80	43	43	40	40	30	30	26	26	12	26	11	26	9	26
2		81	81	80	80	43	43	40	43	30	43	26	43	26	43	26	52
3				81	81	80	80	43	70	43	70	38	70	37	70	35	79
4						81	83	70	83	56	96	43	96	43	96	43	96
5						83		73		69		55		54		52	
6								80		70		70		70		70	
7								83		83		82		81		79	
8										96		96		96		96	

Figure 4.4 State weights produced by $GL(\frac{1}{3})$ for Example 4.2

over $\{1, 2, \ldots, 2^{cn^2}\}$ and the capacity is the total weight of a randomly chosen subset of the items. They presented a polynomial-time algorithm which finds the solution for "almost all" instances with $c > 1$. The result was extended to $c > \frac{1}{2}$ by Frieze (1986).

The probabilistic properties of a "bounded" variant of SSP were investigated by Tinhofer and Schreck (1986).

4.4 COMPUTATIONAL EXPERIMENTS

In the present section we analyse the experimental behaviour of exact and approximate algorithms for SSP on random and deterministic test problems.

The main class of randomly generated test problems we use is

(i) *problems* $P(E)$: w_j uniformly random in $[1, 10^E]$;

$$c = n\,\frac{10^E}{4}.$$

For each pair (n, E), the value of c is such that about half the items can be expected to be in the optimal solution. In all algorithms for SSP, execution is halted as soon as a solution of value c is found. Hence the difficulty of a problem instance is related to the number of different solutions of value c. It follows that problems $P(E)$ tend to be more difficult when E grows. As we will see, truly difficult problems can be obtained only with very high values of 10^E. This confirms, in a sense, a theoretical result obtained by Chvátal (1980), who proved that, for the overwhelming majority of problems $P(n/2)$ (with n large enough), the running time of any algorithm based on branch-and-bound and dynamic programming is proportional at least to $2^{n/10}$. The Chvátal problems, as well as problems $P(E)$ with very high values of E, cannot be generated in practice because of the integer overflow limitation. A class of difficult problems which does not have this drawback is

(ii) *problems EVEN/ODD* : w_j even, uniformly random in $[1, 10^3]$;

$$c = \frac{n10^3}{4} + 1 \quad \text{(odd)}.$$

Since these problems admit no solution of value c, the execution of any enumerative algorithm terminates only after complete exploration of the branch-decision tree. Deterministic problems with the same property have been found by Todd (1980) and Avis (1980):

(iii) *problems TODD* : $w_j = 2^{k+n+1} + 2^{k+j} + 1$, with $k = \lfloor \log_2 n \rfloor$;

$$c = \lfloor 0.5 \textstyle\sum_{j=1}^{n} w_j \rfloor = (n+1)2^{k+n} - 2^k + \left\lfloor \frac{n}{2} \right\rfloor,$$

(iv) *problems AVIS* : $w_j = n(n+1)+j;$

$$c = \left\lfloor \frac{n-1}{2} \right\rfloor n(n+1) + \binom{n}{2}.$$

4.4.1 Exact algorithms

We first compare, on small-size difficult problems, the Fortran IV implementations of the dynamic programming algorithm of Ahrens and Finke (1975) and of algorithm MTSL (Section 4.2.3). We used a CDC-Cyber 730 computer, having 48 bits available for integer operations, in order to be able to work with the large coefficients generated.

Table 4.1 gives the results for problems $P(E)$, with $E = 3, 6, 12$, Table 4.2 those for problems *EVEN/ODD, TODD, AVIS*. Each entry gives the average running

Table 4.1 Problems $P(E)$. CDC-Cyber 730 in seconds. Average times over 10 problems

	$P(3)$: w_j uniformly random in $[1, 10^3]$; $c = n10^3/4$		$P(6)$: w_j uniformly random in $[1, 10^6]$; $c = n10^6/4$		$P(12)$: w_j uniformly random in $[1, 10^{12}]$; $c = n10^{12}/4$	
n	Ahrens and Finke	MTSL	Ahrens and Finke	MTSL	Ahrens and Finke	MTSL
8	0.012	0.004	0.012	0.004	0.013	0.004
12	0.023	0.010	0.029	0.013	0.029	0.013
16	0.040	0.011	0.091	0.049	0.092	0.050
20	0.069	0.007	0.322	0.185	0.422	0.232
24	0.137	0.010	0.640	0.513	2.070	1.098
28	0.349	0.010	1.341	0.647	9.442	6.306
32	0.940	0.009	2.284	0.661	time limit	time limit
36	2.341	0.009	4.268	0.605	—	—
40	5.590	0.011	9.712	0.663	—	—

Table 4.2 Problems *EVEN/ODD*, *TODD*, *AVIS*. CDC-Cyber 730 in seconds

	EVEN/ODD; average times over 10 problems		*TODD*; single trials		*AVIS*; single trials	
n	Ahrens and Finke	MTSL	Ahrens and Finke	MTSL	Ahrens and Finke	MTSL
8	0.013	0.005	0.013	0.002	0.016	0.002
12	0.028	0.021	0.050	0.005	0.041	0.005
16	0.090	0.053	0.199	0.020	0.111	0.012
20	0.392	0.190	0.785	0.257	0.326	0.046
24	1.804	0.525	3.549	0.400	0.815	0.126
28	7.091	0.969	15.741	0.403	2.010	0.291
32	21.961	1.496	70.677	0.407	4.348	0.579
36	time limit	2.184	308.871	0.409	8.345	1.146
40	—	2.941	—	—	15.385	1.780

time, expressed in seconds, computed over 10 problem instances (except for the deterministic *TODD* and *AVIS* problems, for which single runs were executed). Each algorithm had a time limit of 450 seconds to solve the problems generated for each data set. MTSL was always faster than the Ahrens and Finke (1975) algorithm. Table 4.1 shows that problems $P(E)$ become really hard only when very high values of 10^E are employed. Table 4.2 demonstrates that the "artificial" hard problems can still be solved, in reasonable time, by MTSL. (Problems *TODD* cannot be generated for $n \geq 40$ because of integer overflows.)

We used a 32-bit HP 9000/840 computer, having a core memory limitation of 10 Mbytes, to test MTSL on very large "easy" $P(E)$ instances. Since the Fortran implementation of MTSL requires only two vectors of dimension n, we were able to solve problems up to one million variables. Because of integer overflow limitations, the capacity was set to $n10^E/50$, hence E could not be greater than 5. Table 4.3 gives the average times, computed over 20 problem instances, relative to problems $P(2)$, $P(3)$, $P(4)$, $P(5)$. The results show very regular times, growing almost linearly with n. No remarkable difference comes from the different values of E used. The initial value of ϑ ($\vartheta = 45$) always produced the optimal solution. All runs were executed with option "-o" for the Fortran compiler, i.e. with the lowest optimization level.

Table 4.3 Problems $P(E)$. HP 9000/840 in seconds. Average times over 20 problems

n	$P(2)$ w_j uniformly random in $[1, 10^2]$; $c = n10^2/50$	$P(3)$ w_j uniformly random in $[1, 10^3]$; $c = n10^3/50$	$P(4)$ w_j uniformly random in $[1, 10^4]$; $c = n10^4/50$	$P(5)$ w_j uniformly random in $[1, 10^5]$; $c = n10^5/50$
1 000	0.007	0.010	0.022	0.125
2 500	0.009	0.014	0.025	0.116
5 000	0.016	0.020	0.031	0.121
10 000	0.028	0.032	0.046	0.126
25 000	0.070	0.071	0.088	0.173
50 000	0.136	0.138	0.156	0.252
100 000	0.277	0.272	0.295	0.392
250 000	0.691	0.674	0.716	0.801
500 000	1.361	1.360	1.418	1.527
1 000 000	2.696	2.720	2.857	2.948

4.4.2 Approximate algorithms

We used the hard problems of the previous section to experimentally compare approximate algorithms for SSP. The runs were executed on a CDC-Cyber 730 computer, with values of n ranging from 10 to 1 000 for problems *EVEN/ODD* and $P(10)$ ($E = 10$ being the maximum value not producing integer overflows), from 10 to 35 for problems *TODD*. We compared the Fortran IV implementations of the

polynomial-time approximation schemes of Johnson (1974) and Martello and Toth (1984b) and those of the fully polynomial-time approximation schemes of Lawler (1979) and Gens and Levner (1978, 1980) (referred to as J(k), MTSS(k), L(ε) and GL(ε), respectively). The size of the approximate core for MTSS(k) was set to $200/k$.

We used the values $\frac{1}{2}$, $\frac{3}{4}$ and $\frac{6}{7}$ of the worst-case performance ratio r. These are the smallest values which can be imposed on all the schemes. Table 4.4 shows the parameters used and the time and space complexities.

Table 4.4 Time and space complexities

		J(k)		MTSS(k)			L(ε)			GL(ε)	
r	k	time	space	k	time	space	ε	time	space	time	space
$\frac{1}{2}$	1	$O(n)$	$O(n)$	1	$O(n)$	$O(n)$	$\frac{1}{2}$	$O(n+\frac{1}{\varepsilon^3})$	$O(n+\frac{1}{\varepsilon^2})$	$O(\frac{n}{\varepsilon})$	$O(\frac{n}{\varepsilon})$
$\frac{3}{4}$	3	$O(n^3)$	$O(n)$	2	$O(n^2)$	$O(n)$	$\frac{1}{4}$	$O(n+\frac{1}{\varepsilon^3})$	$O(n+\frac{1}{\varepsilon^2})$	$O(\frac{n}{\varepsilon})$	$O(\frac{n}{\varepsilon})$
$\frac{6}{7}$	6	$O(n^6)$	$O(n)$	3	$O(n^3)$	$O(n)$	$\frac{1}{7}$	$O(n+\frac{1}{\varepsilon^3})$	$O(n+\frac{1}{\varepsilon^2})$	$O(\frac{n}{\varepsilon})$	$O(\frac{n}{\varepsilon})$

For each triple (type of problem, value of r, value of n), we generated ten problems and solved them with the four algorithms. The tables give two types of entries: average running times and average percentage errors. The errors were computed with respect to the optimal solution for problems *TODD*. For problems $P(10)$ and *EVEN/ODD* we used the optimal solution when $n < 50$, and the upper bound c (for $P(10)$) or $c-1$ (for *EVEN/ODD*) when $n \geq 50$. When all ten problems were exactly solved by an algorithm, the corresponding error entry is "exact" (entry 0.0000 means that the average percentage error was less than 0.00005).

Table 4.5 gives the results for problems $P(10)$. L(ε) has, in general, very short times and very large errors. This is because the number of large items is very small (for $n \leq 50$) or zero (for $n \geq 100$). MTSS(k) dominates the other algorithms, J(k) dominates GL(ε). For any $n \geq 25$, J(k) gives exactly the same results, independently of r since, for all such cases, set L is empty, so only the greedy algorithm is performed. The running times of GL(ε) grow with n and with r, those of J(k) only with n, those of MTSS(k) only with r (for $n \geq 50$), while L(ε) has an irregular behaviour.

Table 4.6 gives the results for problems *EVEN/ODD*. As in Table 4.5, L(ε) always has very short times and very large errors, MTSS(k) dominates the other algorithms and J(k) dominates GL(ε). The running times and the growing rates of errors are the same as in Table 4.5 while the absolute errors are different. In many cases MTSS(k) found the optimal solution; since, however, the corresponding value does not coincide with c, execution could not stop, so the running times grow with r.

Table 4.7 gives results for problems *TODD*. Since these problems are deterministic, the entries refer to single trials. MTSS(k) dominates all the

Table 4.5 Problems $P(10)$: w_j uniformly random in $[1, 10^{10}]$; $c = n10^{10}/4$. CDC-Cyber 730 in seconds. Average values over 10 problems

		Time				Percentage error			
n	r	J(k)	MTSS(k)	L(ε)	GL(ε)	J(k)	MTSS(k)	L(ε)	GL(ε)
	$\frac{1}{2}$	0.001	0.001	0.004	0.005	2.0871	2.0871	5.5900	2.0307
10	$\frac{3}{4}$	0.001	0.001	0.012	0.009	2.0044	0.4768	3.7928	1.2864
	$\frac{6}{7}$	0.003	0.006	0.025	0.014	0.8909	0.1894	2.8857	0.9088
	$\frac{1}{2}$	0.002	0.003	0.001	0.014	0.3515	0.3515	5.3916	1.8044
25	$\frac{3}{4}$	0.002	0.005	0.008	0.020	0.3515	0.0467	1.9958	0.6695
	$\frac{6}{7}$	0.003	0.035	0.069	0.037	0.3515	0.0049	1.5973	0.6100
	$\frac{1}{2}$	0.004	0.005	0.001	0.029	0.0833	0.0833	0.8870	0.2519
50	$\frac{3}{4}$	0.004	0.009	0.001	0.050	0.0833	0.0058	0.8870	0.1008
	$\frac{6}{7}$	0.004	0.166	0.016	0.079	0.0833	0.0002	0.9902	0.0794
	$\frac{1}{2}$	0.009	0.014	0.002	0.061	0.0082	0.0082	1.0991	0.0611
100	$\frac{3}{4}$	0.009	0.020	0.001	0.093	0.0082	0.0004	1.0991	0.0708
	$\frac{6}{7}$	0.010	0.207	0.001	0.157	0.0082	0.0001	1.0991	0.0541
	$\frac{1}{2}$	0.020	0.022	0.003	0.112	0.0032	0.0039	0.7441	0.0077
250	$\frac{3}{4}$	0.022	0.029	0.003	0.235	0.0032	0.0004	0.7441	0.0070
	$\frac{6}{7}$	0.022	0.158	0.003	0.374	0.0032	0.0000	0.7441	0.0059
	$\frac{1}{2}$	0.049	0.022	0.008	0.254	0.0010	0.0040	0.2890	0.0016
500	$\frac{3}{4}$	0.042	0.033	0.006	0.438	0.0010	0.0001	0.2890	0.0016
	$\frac{6}{7}$	0.047	0.180	0.007	0.685	0.0010	0.0000	0.2890	0.0015
	$\frac{1}{2}$	0.100	0.024	0.013	0.540	0.0002	0.0014	0.1954	0.0005
1000	$\frac{3}{4}$	0.102	0.030	0.013	0.909	0.0002	0.0001	0.1954	0.0006
	$\frac{6}{7}$	0.102	0.224	0.013	1.374	0.0002	0.0000	0.1954	0.0005

Table 4.6 Problems *EVEN/ODD*. CDC-Cyber 730 in seconds. Average times over 10 problems

		Time				Percentage error			
n	r	J(k)	MTSS(k)	L(ε)	GL(ε)	J(k)	MTSS(k)	L(ε)	GL(ε)
	$\frac{1}{2}$	0.001	0.001	0.005	0.005	2.2649	2.2649	7.5859	1.5131
10	$\frac{3}{4}$	0.001	0.002	0.012	0.009	2.3209	0.8325	3.1369	0.8403
	$\frac{6}{7}$	0.003	0.007	0.025	0.013	0.9202	0.0720	2.5209	0.9041
	$\frac{1}{2}$	0.002	0.003	0.001	0.015	0.2432	0.2432	7.6416	1.0688
25	$\frac{3}{4}$	0.002	0.005	0.011	0.026	0.2432	0.0384	2.3360	0.4288
	$\frac{6}{7}$	0.002	0.048	0.077	0.042	0.2432	exact	1.9808	0.3584
	$\frac{1}{2}$	0.004	0.006	0.001	0.030	0.0400	0.0400	2.4480	0.1424
50	$\frac{3}{4}$	0.004	0.011	0.001	0.051	0.0400	0.0016	2.4480	0.1680
	$\frac{6}{7}$	0.004	0.287	0.019	0.084	0.0400	exact	1.1232	0.0816
	$\frac{1}{2}$	0.009	0.011	0.002	0.060	0.0160	0.0160	0.7352	0.0792
100	$\frac{3}{4}$	0.009	0.028	0.002	0.100	0.0160	exact	0.7352	0.0792
	$\frac{6}{7}$	0.008	0.274	0.002	0.166	0.0160	exact	0.7352	0.0520
	$\frac{1}{2}$	0.022	0.022	0.003	0.141	0.0019	0.0006	0.4221	0.0080
250	$\frac{3}{4}$	0.021	0.028	0.004	0.235	0.0019	exact	0.4221	0.0070
	$\frac{6}{7}$	0.021	0.242	0.003	0.380	0.0019	exact	0.4221	0.0058
	$\frac{1}{2}$	0.047	0.026	0.007	0.291	0.0003	0.0006	0.2682	0.0021
500	$\frac{3}{4}$	0.047	0.031	0.007	0.483	0.0003	exact	0.2682	0.0021
	$\frac{6}{7}$	0.047	0.257	0.007	0.774	0.0003	exact	0.2682	0.0024
	$\frac{1}{2}$	0.104	0.029	0.013	0.595	exact	0.0010	0.1325	0.0002
1000	$\frac{3}{4}$	0.104	0.033	0.014	0.992	exact	exact	0.1325	0.0002
	$\frac{6}{7}$	0.104	0.293	0.014	1.567	exact	exact	0.1325	0.0001

Table 4.7 Problems *TODD*. CDC-Cyber 730 in seconds. Single trials

		Time				Percentage error			
n	r	J(k)	MTSS(k)	L(ε)	GL(ε)	J(k)	MTSS(k)	L(ε)	GL(ε)
	$\frac{1}{2}$	0.001	0.001	0.002	0.005	9.9721	9.9721	8.2795	exact
10	$\frac{3}{4}$	0.001	0.002	0.013	0.008	9.9721	exact	4.4157	exact
	$\frac{6}{7}$	0.004	0.006	0.022	0.011	exact	exact	2.1366	exact
	$\frac{1}{2}$	0.001	0.001	0.001	0.008	exact	exact	12.3660	6.1343
15	$\frac{3}{4}$	0.002	0.002	0.016	0.016	exact	exact	6.2072	3.0185
	$\frac{6}{7}$	0.001	0.012	0.033	0.023	exact	exact	1.4548	exact
	$\frac{1}{2}$	0.001	0.003	0.001	0.014	4.7761	4.7761	4.7482	4.7482
20	$\frac{3}{4}$	0.001	0.003	0.006	0.023	4.7761	exact	4.7482	exact
	$\frac{6}{7}$	0.002	0.018	0.036	0.040	4.7761	exact	2.3787	exact
	$\frac{1}{2}$	0.001	0.003	0.001	0.016	exact	exact	7.6896	7.6896
25	$\frac{3}{4}$	0.001	0.004	0.001	0.036	exact	exact	7.6896	exact
	$\frac{6}{7}$	0.001	0.028	0.048	0.059	exact	exact	3.3638	1.9210
	$\frac{1}{2}$	0.002	0.003	0.001	0.017	3.2261	3.2261	3.2255	3.2255
30	$\frac{3}{4}$	0.003	0.005	0.001	0.030	3.2261	exact	3.2255	3.2253
	$\frac{6}{7}$	0.002	0.038	0.017	0.052	3.2261	exact	0.8063	exact
	$\frac{1}{2}$	0.003	0.004	0.001	0.021	exact	exact	5.5555	2.7777
35	$\frac{3}{4}$	0.002	0.004	0.001	0.035	exact	exact	5.5555	2.7777
	$\frac{6}{7}$	0.002	0.045	0.015	0.062	exact	exact	1.3888	exact

algorithms, while L(ε) is generally dominated by all the algorithms. J(k) dominates GL(ε) for n odd (J(1) always finds the optimal solution). For n even, GL(ε) has higher times but much smaller errors than J(k), MTSS(2) always finds the optimal solutions, MTSS(1) only for n odd. This behaviour of the algorithms can be explained by analysing the structure of the optimal solution to problems *TODD*. Let $m = \lfloor n/2 \rfloor$, so $c = (n+1)2^{k+n} - 2^k + m$. Hence the number of items in any feasible solution is at most m since, by algebraic manipulation, the sum of the $m+1$ smallest weights is

$$\sum_{i=1}^{m+1} w_i = 2(m+1)2^{k+n} + 2^{k+1}(2^{m+1} - 1) + (m+1) > c$$

(in problems *TODD* the w_i's are given for increasing values). For n odd ($n = 2m + 1$), the sum of the m largest weights is feasible, since

$$\sum_{i=n-m+1}^{n} w_i = (2m+2)2^{k+n} - 2^{k+n-m+1} + m < c,$$

and hence optimal. So, after sorting, the greedy algorithm (J(1) or MTSS(1)) finds the optimal solution. For n even ($n = 2m$), (a) any solution including w_n includes at most $m - 2$ further items, since

$$w_n + \sum_{i=1}^{m-1} w_i = (2m+1)2^{k+n} + 2^k(2^m - 2) + m > c;$$

(b) it follows that the best solution including w_n has value

$$z_1 = \sum_{i=n-m+2}^{n} w_i = 2m2^{k+n} - 2^{k+n-m+2} + m - 1 < c;$$

(c) the best solution not including w_n has value

$$z_2 = \sum_{i=n-m}^{n-1} w_i = (2m+1)2^{k+n} - 2^{k+n-m} + m < c,$$

and $z_2 > z_1$. So z_2 is the optimal solution value and MTSS(2) finds it when, after sorting, it applies the greedy algorithm starting from the second element.

We do not give the results for problems *AVIS*, for which the algorithms have a behaviour similar to that of problems *TODD*. In fact, let $s = \lfloor (n-1)/2 \rfloor$, so $c = sn(n+1) + n(n-1)/2$. Since the sum of the $s+1$ smallest weights is

$$\sum_{i=1}^{s+1} w_i = sn(n+1) + n(n+1) + \tfrac{1}{2}(s+1)(s+2) > c,$$

any feasible solution will include, at most, s items. The sum of the s largest weights is feasible, since

$$\sum_{i=n-s+1}^{n} w_i = sn(n+1) + s(n-s) + \tfrac{1}{2}s(s+1) \leq c,$$

hence optimal. So, after sorting, the greedy algorithms $J(1)$ and MTSS(1) always find the optimal solution.

The computational results of this section (and others, reported in Martello and Toth (1985a)) show that all the polynomial-time approximation schemes for SSP

have an average performance much better than their worst-case performance. So, in practical applications, we can obtain good results with short computing times, i.e. by imposing small values of the worst-case performance ratio.

Although polynomial-time approximation schemes have a worse bound on computing time, their average performance appears superior to that of the fully polynomial-time approximation schemes, in the sense that they generally give better results with shorter computing times and fewer core memory requirements.

The most efficient scheme is MTSS(k). For $n \geq 50$, the largest average error of MTSS(2) was 0.0075 per cent, that of MTSS(3) 0.0005 per cent. So, for practical purposes, one of these two algorithms should be selected while using higher values of k would probably be useless.