Dynamic programming on the word RAM

David Pisinger

Abstract

Dynamic programming is one of the fundamental techniques for solving optimization problems. In the present paper we propose a general framework which can be used to decrease the time and space complexity of these algorithms with a logarithmic factor. The framework is based on word encoding, i.e. by representing subsolutions as bits in an integer. In this way word parallelism can be used in the evaluation of the dynamic programming recursion. Using this encoding the subset-sum problem can be solved in $O(nb/\log b)$ time and $O(b/\log b)$ space, where n is the number of integers given and b is the target sum. The knapsack problem can be solved in $O(nb/\log b + nz/\log z)$ time and $O(b/\log b + z/\log z)$ space where n is the number of items, b is the capacity and z is the optimal solution value. The problem of finding a path of a given length b in a layered network $G = (V, E)$ can be solved in $O(|E|b/\log b)$ time and $O(|V|b/\log b)$ space. Several other examples are given showing the generality of the achieved technique. Extensive computational experiments are provided to demonstrate that the achieved results are not only of theoretical interest but actually lead to algorithms which are up to two orders of magnitude faster than their predecessors. This is a surprising observation as the increase in speed is larger than the word size of the processor.

1 Introduction

An optimization problem exhibits the property of *optimal substructure* if an optimal solution to the problem consists of a sequence of optimal solutions to subproblems. Such problems may be solved through *dynamic programming* as this framework takes advantage of overlapping subproblems to decrease the computational effort [4]. Applying dynamic programming to $\mathcal{NP}$-hard optimization problems may lead to algorithms with pseudopolynomial running time, which with respect to the worst-case complexity are to be prefered to branch-and-bound techniques. Moreover, dynamic programming algorithms generally have the additional benefit that we do not only obtain a single solution but a whole table of optimal subsolutions corresponding to different values of the constraints. Thus e.g. using dynamic programming to solve a path problem in a layered network gives a table of all achievable path-lengths. This additional information may be very valuable to the decision maker, thus motivating the use of dynamic programming in favor of other enumerative approaches. Solving the problem for all values of the constraints will change the problems considered to polynomial problems with respect to the input and output size. For a general introduction to dynamic programming see the text book by Ibaraki [11].

Algorithms exploiting word-parallelism have been studied in the last years within the field of algorithms dealing with polynomial problems. For an excellent survey on sorting and searching algorithms on the word RAM see [9]. These algorithms have improved the time complexity of several problems and given a new insight into the design of algorithms. Unfortunately the improved time complexities seldom lead to practical improvements of the performance, as they are beaten by algorithms with theoretically worse time complexity but where the constants in the complexity term are much smaller. The present paper exploits word-parallelism in dynamic programming algorithms for $\mathcal{NP}$-hard optimization problems. As the computational effort for solving these problems is huge, word parallelism is of great importance both from a theoretical and practical aspect.

The basic techniques are first introduced for the subset-sum problem. As the coefficients in the objective function and in the constraint are the same for this problem a simple representation can be achieved, where each bit indicates whether a subset-sum can be achieved with a sum which equals the bit position. In this way we may exploit word parallelism to decrease the running time of the dynamic programming algorithm to $O(nb/\log b)$ where n is the number of integers given and b is the capacity. The following section deals with the knapsack problem which is more complicated due to the fact that the coefficients in the objective function and in the constraint may differ. To exploit word-parallelism a result table is constructed which returns the solution word corresponding to two words of input. The result table may however become unacceptable large, so a reduced result table is constructed which only accepts one tenth of a machine word as argument. Using this reduced result table it is possible to decrease the space bound at the expense of a constant increase in the time complexity. In this way the time bound becomes $O(nm/\log m)$ with space consumption $O(m/\log m)$, where n is the number of items, b is the capacity, z is the optimal solution value and $m = \max\{b, z\}$.

With the two prototypes at hand — subset-sum and knapsack problem — we can generalize the techniques to numerous problems. First, we show in Section 4 how a path of a given length b in a layered network can be determined in $O(|E|b/\log b)$ time and $O(|V|b/\log b)$ space. This problem is further generalized to the case where an additional constraint is imposed on the derived path. This problem can be solved in $O(|E|m/\log m)$ time and $O(|V|m/\log m)$ space where $m = \max\{b, d\}$, and d is the bound on the additional constraint.

In the final section we apply the techniques to develop new and more efficient dynamic programming algorithms for several problems. These problems include the change-making problem, two-partition problem, and several bounded and unbounded versions of the knapsack problem.

The paper is concluded with a computational study of the presented subset-sum algorithm which demonstrates that the achieved results are not only of theoretical interest, but that the techniques developed actually may lead to algorithms which are up to two orders of magnitude faster than their predecessors.

We will assume that the largest coefficient in the input and output data is m and that the word size of the computer is w, where $w \geq \log m$ as otherwise we cannot store the coefficient. When deriving the time and space complexity of the presented algorithms we will assume that

$$w = \Theta(\log m) \tag{1}$$

The model of computation is the *word RAM* [9] thus we can e.g. perform binary `and`, binary `or`, and bitwise shift with w bits in constant time on words of size w. None of the developed algorithms make use of multiplication apart from the process of indexing multi-dimensional tables. The indexing can however be implemented by a single shift operation per index if the domain size of the index is extended to the nearest larger power of two. Thus multiplication is not strictly needed, and hence only AC^0 instructions are used.

The terminology used in this paper is as follows: All logarithms are base-two. All bits in a word are numbered from left to right $0, 1, \ldots, w-1$ and words are numbered in a similar way. For a word size of $w = 8$ this can be illustrated as:

word no.	0	1	2	3	$\cdots$
bit no.	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	$\cdots$

2 Subset-sum problem

We will introduce the basic techniques by considering the subset-sum problem: Given a set $S = \{a_1, a_2, \ldots, a_n\}$ of nonnegative integers and an additional integer b, the subset-sum problem asks to choose a subset $\overline{S}$ of S such that $\sum_{\overline{S}} a_i$ is maximized without exceeding b. The integers a_i are frequently denoted the *weights* and b the *capacity*.

The problem is $\mathcal{NP}$-hard, but it can be solved through dynamic programming in time $O(nb)$ through a straightforward recursion by Bellman [2]. Let $f_{i,j}$ for $i = 0, \ldots, n$, $j = 0, \ldots, b$ be a solution to the subset-sum problem defined on items $S_i = \{a_1, \ldots, a_i\}$ with $b = j$. To initialize the recursion we set $f_{0,j} = 0$ for $j = 0, \ldots, b$. Subsequent values of f are derived as

$$f_{i,j} = \max\{f_{i-1,j}, f_{i-1,j-a_i} + a_i\} \tag{2}$$

for $i = 1, \ldots, n$ and $j = 0, \ldots, b$. The optimal solution value is given by $f_{n,b}$. Since the table $f_{i,j}$ has size $O(nb)$ and each entry can be derived in constant time, the time and space complexity becomes $O(nb)$.

Notice that dynamic programming in this case not only solves the problem for the given capacity b but for all capacities $0, \ldots, b$. This variant of the problem appears in the industry where the capacity is not known exactly in advance but where the choice depends on the solutions achievable [1]. The table $f_{n,*}$ provides the decision maker with a visual picture of the solution value as function of the capacity. Also, when solving certain bin-packing problems through branch-and-price the subset-sum problem and related problems appear as the pricing problem. Here the table $f_{n,*}$ contains information about several other solutions which can be used to add more than one column to the master problem thus speeding up the solution process. As the branching part of the branch-and-price algorithm may have forbidden certain solutions, the table of alternative solutions $f_{n,*}$ can be used to choose the second-best choice.

Other approaches for the subset-sum problem include branch-and-bound [15] and balanced dynamic programming [19]. The first approaches run in exponential time while the second has the time and space complexity $O(n \max_{i \in S} a_i)$. Although these approaches frequently are able to solve the subset-sum problem faster than the Bellman recursion, they only return the solution value for a single capacity value.

The Bellman recursion for subset-sum is suitable for word encoding. Let $g_{i,j} = 1$ iff there exists a subset of the weights $S_i = \{a_1, \ldots, a_i\}$ which sums to j. In this way, $g_{0,j} = 0$ for $j = 0, \ldots, b$ and we may use the recursion

$$g_{i,j} = 1 \quad \text{iff} \quad (g_{i-1,j} = 1 \text{ or } g_{i-1,j-a_i} = 1) \tag{3}$$

for $i = 1, \ldots, n$ and $j = 0, \ldots, b$.

Proposition 1 *The table $g_{n,*}$ of optimal solution values can be derived in $O(nb/\log b)$ time and $O(b/\log b)$ space using recursion (3).*

Proof As we only need one bit for representing each entry in $g_{i,j}$ we may use word encoding to store w entries in each integer word. This means that the list $g_{i,*}$ takes up $O(b/\log b)$ space. To derive $g_{i,*}$ from $g_{i-1,*}$ we simply copy $g_{i-1,*}$ to a new table $g'_{i-1,*}$ which is shifted right by a_i bits. Then we merge the two lists $g_{i-1,*}, g'_{i-1,*}$ through binary `or` obtaining $g_{i,*}$. The whole operation can be done in $O(b/\log b)$ time as each word can be shifted in constant time (Notice that if $a_i > w$ then we first shift the table by $\lfloor a_i/w \rfloor$ whole words, followed by a shift of $a_i - \lfloor a_i/w \rfloor$ bits. Assuming that the word size w is a power of two, $\lfloor a_i/w \rfloor$ can be implemented by a simple shift operations), and the binary `or` of two words also can be done in constant time. This proves the time complexity.

The space complexity appears by observing that we only need $g_{i-1,*}$ and $g'_{i-1,*}$ to derive $g_{i,*}$, thus only three arrays are needed. Actually, the whole operation can be done in-place in one single array by considering the values j in decreasing order $b, \ldots, 0$.

The optimal solution value z is found as the largest value $j \leq b$ for which $g_{n,j} = 1$. This can easily be done in $O(b/\log b)$ time as we start from the right-most word, and repeatedly move leftward skipping whole words as long as they have the value zero. When a word different from zero is identified, we simply use linear search to find the right-most bit. The whole operation takes $O(b/\log b + \log w) = O(b/\log b)$. ■

Proposition 2 *For a given solution value z the corresponding subset $\overline{S}$ of S which satisfies that $\sum_{\overline{S}} a_i = z$ can be derived in $O(nb/\log b)$ time and $O(b/\log b)$ space, i.e. the same complexity as solving the recursion (3).*

Proof We may assume that $z = b$ as we simply can solve the problem once more with capacity equal to z. To find $\overline{S}$ we may apply the framework presented by Pferschy [17]: The set S is divided into two equally sized parts S_1 and S_2. Without loss of generality we may assume that $|S|$ is even, and thus $|S_1| = |S_2| = n/2$. Each of the problems is now solved separately returning two tables of optimal solution values $g_{n/2,*}$ and $g'_{n/2,*}$. The first table $g_{n/2,*}$ is obtained in the "ordinary" way, while the second table $g'_{n/2,*}$ is obtained in a "reverse" way, i.e. we start with $g'_{0,b} = 1$ and perform left shifts where we previously performed right shifts. The two tables $g_{n/2,*}$ and $g'_{n/2,*}$ are `and`'ed together giving a new table h_*. If $h_j = 0$ for all $j = 0, \ldots, b$ then it is not possible to obtain the solution value b. Otherwise assume that $h_j = 1$ for a specific value of j meaning that $g_{n/2,j} = g'_{n/2,b-j} = 1$. Hence to obtain the sum b we must find a subset of S_1 which sums to j and a subset of S_2 which sums to $b - j$. The same approach is used recursively until each subset consists of a single weight, in which case it is trivial to determine whether this weight should be in $\overline{S}$. The total complexity becomes

$$O(2\frac{n}{2}b/\log b) + O(2\frac{n}{4}j/\log b + 2\frac{n}{4}(b-j)/\log b) + \cdots$$

This can be reduced to

$$O(nb/\log b) + O(\frac{n}{2}b/\log b) + O(\frac{n}{4}b/\log b) + \cdots$$

which sums to $O(nb/\log b)$ as $\sum_{i=1}^{n} \frac{1}{i} \leq 2$. ■

3 Knapsack problem

Given a set S of nonnegative integer pairs $\{(a_1, c_1), (a_2, c_2), \ldots, (a_n, c_n)\}$ and an additional integer b. The knapsack problem asks to choose a subset $\overline{S}$ of S such that $\sum_{\overline{S}} c_i$ is maximized subject to the constraint $\sum_{\overline{S}} a_i \leq b$. The integers c_i are denoted the *profits*, a_i are the *weights* and b is the *capacity*.

The knapsack problem is well studied in the literature. Several branch-and-bound algorithms have been presented in [15] and a balanced dynamic programming recursion with time and space complexity $O(n \max_{i \in S} a_i \max_{i \in S} c_i)$ was presented in [19]. For a recent survey on knapsack problems see [21]. All the described algorithms however only solve the problem for a single value of the capacity.

If we want to solve the problem for all capacities $0, \ldots, b$ the straightforward recursion by Bellman [2] may be used. Let $f_{i,j}$ for $i = 0, \ldots, n$, $j = 0, \ldots, b$ be a solution to the knapsack problem defined on $S_i = \{(a_1, c_1), (a_2, c_2), \ldots, (a_i, c_i)\}$ and with restricted capacity $b = j$. To initialize the recursion we set $f_{0,j} = 0$ for $j = 0, \ldots, b$. For the following values of f we use the recursion

$$f_{i,j} = \max\{f_{i-1,j}, f_{i-1,j-a_i} + c_i\} \tag{4}$$

for $i = 1, \ldots, n$ and $j = 0, \ldots, b$. The optimal solution value is found as $z = f_{n,b}$. The time and space complexity of this algorithm is obviously $O(nb)$.

To obtain a more compact representation based on word encoding we need to define a *state*. For a given value of i and for each $u = 0, \ldots, b$ we obtain a state as the pair (u, v) defined by $f_{i,u} = v$. If we have two states (u', v') and (u'', v'') where $u' < u''$ and $v' \geq v''$ then state (u', v') *dominates* state (u'', v''). It is obvious that a dominated state never will lead to an optimal solution and thus can be discarded.

Example 1 *We have* $S = \{(a_1, c_1), (a_2, c_2)\} = \{(3,1), (1,3)\}$ *and* $b = 4$. *The left table illustrates the Bellman recursion while the right table shows the corresponding list of undominated states. Thus e.g.* $f_{2,1} = 3$ *means that a profit sum of* 3 *can be obtained for the capacity* 1. *In the list of undominated states* $\{(0,0), (1,3), (4,4)\}$ *for* $i = 2$, *state* $(1,3)$ *means that a profit sum of* 3 *can be obtained with a capacity of* 1.

		j				
	$f_{i,j}$	0	1	2	3	4
	0	0	0	0	0	0
i	1	0	0	0	1	1
	2	0	3	3	3	4

	states	(u,v)
	0	(0,0)
i	1	(0,0) (3,1)
	2	(0,0) (1,3) (4,4)

Now we can adapt word encoding to the knapsack problems as follows. We introduce two binary tables $g_{i,j}$ and $h_{i,j}$ for $i = 0, \ldots, n$ and $j = 0, \ldots, b$. Here we have $g_{i,u} = 1$ and $h_{i,v} = 1$ iff an undominated state (u, v) exists in $f_{i,*}$. Thus g is a table of the undominated weight sums and h is a table of the undominated profit sums. As we only need one bit to express the value of g and h we may reduce the space complexity by storing w entries in each word. The encoded representation $g_{i,j}, h_{i,j}$ can be transformed on-line to the ordinary representation $f_{i,j}$ in linear time (and vice-versa). Hence, using the Bellman recursion (4) on the compact representation with on-line conversion of the data immediately leads to the improved space complexity $O(nm/\log m)$ where $m = \max\{b, z\}$. As z is not known in advance one may use the Dantzig upper bound [5] for bounding z, noting that the Dantzig bound is at most twice as large as the optimal solution.

The relation between the two tables $g_{i,j}$ and $h_{i,j}$ and the original table $f_{i,j}$ is defined as follows:

Observation 1 *If the k'th 1-bit in* $g_{i,j}$ *is obtained for* $j = u$ *and the k'th 1-bit in* $h_{i,j}$ *is obtained for* $j = v$ *then* $f_{i,u} = v$.

In other words, if the k'th 1-bits in the two tables form the state (u, v) then it means that a profit sum of v can be obtained for the capacity u.

To also decrease the time complexity we first need to adapt the Bellman recursion to the compact representation. An obvious idea is to use the same approach as for the subset-sum problem independently on the weights and profits (i.e. tables g and h) which however is not a feasible approach as illustrated by the following example.

Example 2 *Using the same instance as in Example 1 we get*

$g_{i,j}$	0	1	2	3	4
0	1	0	0	0	0
1	1	0	0	1	0
2	1	1	0	1	1

$h_{i,j}$	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	1	0	1	1

As $g_{2,1} = 1$ *is the second 1-bit in table* $g_{2,*}$ *and* $h_{2,1} = 1$ *is the second 1-bit in table* $h_{2,*}$ *we should demand that* $f_{2,1} = 1$. *However, this is not the case.*

The example shows the fact that the recursion must take both tables g and h into account. Thus to derive tables $g_{i,*}$ and $h_{i,*}$ from $g_{i-1,*}$ and $h_{i-1,*}$ we use the following approach. First we copy $g_{i-1,*}$ to a new table $g'_{i-1,*}$ which is shifted right by a_i bits. In a similar way we copy $h_{i-1,*}$ to $h'_{i-1,*}$ shifted right by c_i bits. Now we need to merge the tables f, f', g, g' in a way which respects Observation 1. For this purpose we introduce the following variables

k_g	is the number of 1-bits met in table $g_{i-1,*}$.
k'_g	is the number of 1-bits met in table $g'_{i-1,*}$.
k_h	is the number of 1-bits met in table $h_{i-1,*}$.
k'_h	is the number of 1-bits met in table $h'_{i-1,*}$.
p_g	is the current position in tables $g_{i-1,*}$ and $g'_{i-1,*}$.
p_h	is the current position in tables $h_{i-1,*}$ and $h'_{i-1,*}$.

To simplify the following algorithm we introduce a procedure `emit`(e_g, e_h) which outputs the values e_g, e_h, increments p_g or p_h and adjusts k_g, k'_g, k_h, k'_h accordingly.

`procedure emit`(e_g, e_h)
`if` $e_g \neq \perp$ `then write` e_g `to` g_{i,p_g} ; $p_g \leftarrow p_g + 1; k_g \leftarrow k_g + g_{i-1,p_g}; k'_g \leftarrow k'_g + g'_{i-1,p_g}$
`if` $e_h \neq \perp$ `then write` e_h `to` h_{i,p_h} ; $p_h \leftarrow p_h + 1; k_h \leftarrow k_h + h_{i-1,p_h}; k'_h \leftarrow k'_h + h'_{i-1,p_h}$

The main algorithm may now be described as follows. All references are to $g_{i-1,*}$ or $h_{i-1,*}$ thus the first index will be omitted to simplify the notation.

0 `procedure merge`
`initialize` $k_g, k'_g, k_h, k'_h, p_g, p_h \leftarrow 0$.
1 `if` $g_{p_g} = 0$ `and` $g'_{p_g} = 0$ `then emit`$(0, \perp)$ `goto 1`
2 `if` $g_{p_g} = 1$ `and` $k_h < k_g$ `then emit`$(\perp, 0)$ `goto 1`
`if` $g'_{p_g} = 1$ `and` $k'_h < k'_g$ `then emit`$(\perp, 0)$ `goto 1`
3 `if` $g_{p_g} = 1$ `and` $h_{p_h} = 1$ `and` $k_h = k_g$ `then emit`$(1, 1)$ `goto 1`
`if` $g'_{p_g} = 1$ `and` $h'_{p_h} = 1$ `and` $k'_h = k'_g$ `then emit`$(1, 1)$ `goto 1`
4 `if` $g_{p_g} = 1$ `and` $k_h \geq k_g$ `then emit`$(0, \perp)$ `goto 1`
`if` $g'_{p_g} = 1$ `and` $k'_h \geq k'_g$ `then emit`$(0, \perp)$ `goto 1`

To simplify the algorithm the stop criteria has been omitted, as it is a trivial task to ensure a proper termination.

Example 3 *Using algorithm* `merge` *for the instance from Example 2 we get the following tables*

$g_{i,j}$	0	1	2	3	4
0	1	0	0	0	0
1	1	0	0	1	0
2	1	1	0	0	1

$h_{i,j}$	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	0	0	1	1

To illustrate the principle the following tables show how $g_{2,*}$ and $h_{2,*}$ is obtained:

	0	*1*	*2*	*3*	*4*
$g_{1,*}$	*1*	*0*	*0*	*1*	*0*
$g'_{1,*}$	*0*	*1*	*0*	*0*	*1*

	0	*1*	*2*	*3*	*4*
$h_{1,*}$	*1*	*1*	*0*	*0*	*0*
$h'_{1,*}$	*0*	*0*	*0*	*1*	*1*

p_g	p_h	*step*	*action*
0	*0*	*3a*	`emit`$(1,1)$
1	*1*	*2b*	`emit`$(\perp,0)$
1	*2*	*2b*	`emit`$(\perp,0)$
1	*3*	*3b*	`emit`$(1,1)$
2	*4*	*1*	`emit`$(\perp,0)$
3	*4*	*4a*	`emit`$(0,\perp)$
4	*4*	*3b*	`emit`$(1,1)$

The table $g'_{1,*}$ is obtained by shifting $g_{1,*}$ right by $a_2 = 1$ while table $h'_{1,*}$ is obtained by shifting $h_{1,*}$ right by $c_2 = 3$. The right-most table shows each step of algorithm `merge` *and the corresponding emitted values.*

Proposition 3 *Algorithm* `merge` *correctly merges the tables g^1, h^1, g^2, h^2 in time $O(m)$ where $m = \max\{b, z\}$.*

Proof To start with the time complexity, we notice that each iteration of the main loop emits at least one bit and thus increments either p_g or p_h. As $p_g \leq b$ and $p_h \leq z$ we immediately get the stated bound.

To prove the correctness we should recapitulate the definition of states in Observation 1. Thus step 1 is searching forward in the weight sum table g or g' until the weight-part of the next state is found. In steps 2–4 we have identified this weight-part of the next state, and for symmetry reasons we may assume that $g_{p_g} = 1$. Now three possibilities exist:

$k_h < k_g$ (**step 2**) We have met more bits in table g than in table h thus we need to search forward to find the corresponding entry in the h table to define a state (p_g, β) where $\beta > p_h$. Thus it seems safe to move forward in table h, but there is a hatch: It could be that (p_g, p_h) defined a state in table g', h'. If this was the case, the state (p_g, p_h) would be dominated by state (p_g, β) and thus it is indeed safe to move forward in table h.

$k_h = k_g$ (**step 3**) In this case we have met a similar amount of bits in tables g and h thus if $g_{p_g} = h_{p_h} = 1$ then the pair (p_g, p_h) defines a state. The state (p_g, p_h) cannot be dominated since this was already checked in step 2.

$k_h > k_g$ (**step 4**) In this case we have met more bits in table h than in table g thus we need to increase the position in table g to find a pair (γ, p_h) which make a state. As all cases of dominance have been tested in the previous steps it is safe to move forward in table g.

■

As all decision are based on the truth value of $\mathrm{sign}(k_h - k_g)$, $\mathrm{sign}(k'_h - k'_g)$ and the bits g_{i-1,p_g}, g'_{i-1,p_g}, h_{i-1,p_h}, h'_{i-1,p_h}, the whole process can be controled by a single action table. Each step of algorithm `merge` consists of a single lookup in the action table which has dimensions $3 \times 3 \times 2 \times 2 \times 2 \times 2 = (2+1)^2 2^4$. The output of the table will consist of a pair of bits stating the emitted values (e_g, e_h).

Assume that a new action table was obtained as a product of α times the original action table. In this way we would be able to process up to α bits in one single table lookup, thus getting an improved time and space complexity.

Proposition 4 *Algorithm* `merge` *controled by an action table for* $\alpha = w/10$ *bits runs in* $O(m/\log m)$ *time and space where* $m = \max\{b, z\}$.

Proof The correctness of the algorithm is obvious as the new action table is the product of α of the original action tables.

To prove the time complexity we notice that in each iteration we are able to process α bits, thus the time complexity is $O(m/\alpha)$. Due to assumption (1) we know that the word size corresponds to the size of the coefficients, i.e. that $w = \Theta(\log m)$ and thus the running time is

$$O(m/\alpha) = O(m/w) = O(m/\log m)$$

The space complexity is obtained as follows: The action depends on the difference between the number of bits met in table g and h i.e. whether

$$k_h - k_g < -\alpha, \qquad k_h - k_g = \{-\alpha + 1, \ldots, \alpha - 1\}, \qquad k_h - k_g > \alpha$$

thus the number of distinct values is $2\alpha + 1$. The same applies for the value of $k'_h - k'_g$. Moreover, the action depends on the entries

$$g_{i-1,p_g}, g_{i-1,p_g+1}, \ldots, g_{i-1,p_g+\alpha-1}$$

which can have 2^α different values. The same applies for g'_{i-1,p_g}, h_{i-1,p_h} and h'_{i-1,p_h}. In total the action table will have size $(2\alpha+1)^2 2^{4\alpha}$ i.e. it will use $O(\alpha^2 2^{4\alpha})$ space. Now since $O(\alpha^2) \leq O(2^\alpha)$ and since $\alpha = w/10$ we get the space complexity

$$O(\alpha^2 2^{4\alpha}) \leq O(2^{5\alpha}) = O(2^{w/2})$$

The space consumption depends on the size of the action table and of the dynamic programming table. Due to assumption (1) we have $w = \Theta(\log m)$ and thus the space complexity becomes

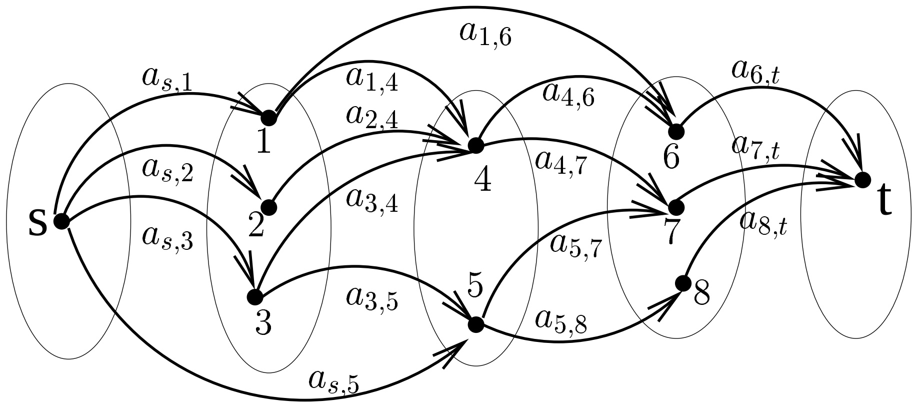
$$O(2^{w/2} + m/\log m) = O(\sqrt{2^w} + m/\log m) = O(\sqrt{m} + m/\log m) = O(m/\log m)$$

■

Corollary 1 *The knapsack problem can be solved in* $O(nm/\log m)$ *time and* $O(m/\log m)$ *space where* $m = \max\{b, z\}$.

Proof The stated follows immediately by using algorithm `merge` for all items $1, \ldots, n$. ■

If we for a given solution value z wish to find the subset $\overline{S}$ which satisfies $\sum_{\overline{S}} c_i = z$ then we may use the same approach as in Proposition 2.



4 Path in a layered network

Given a layered network $G = (V, E)$ with *source* s and *terminal* t. Each edge (i, j) has an associated nonnegative integer weight a_{ij} called the *distance*. We wish to answer the question; does there exist a path of length b from s to t. This problem is $\mathcal{NP}$-hard as can be proved by reduction from subset-sum.

The problem has several applications in branch-and-price algorithms for the Vehicle Routing Problem, where the pricing problem may be seen as a shortest path problem [6]. Since the branch-and-bound search may have forbidden the shortest path from s to t it becomes necessary to find paths of alternative lengths.

In order to use word encoding we introduce the binary table $f_{i,j}$ for $i \in V$, $j = 0, \ldots, b$. Here $f_{i,j} = 1$ iff a path of length j exists from s to i. Using the same arguments as for the subset-sum problem we notice that the table can be stored in $O(|V|b/\log b)$ space. For node s we initialize the table as $f_{s,j} = 0$ for $j = 1, \ldots, b$ and $f_{s,0} = 1$. Successive values of $f_{i,*}$ are obtained by considering the nodes one layer at a time, using the same algorithm as for the subset-sum problem. In this way $f_{i,*}$ is obtained by `or`'ing the tables $f_{h,*}$ (shifted right by $a_{h,i}$ bits) for all ingoing edges (h, i). The total running time becomes $O(|E|b/\log b)$. If the graph is sparse, i.e. $|E| = O(|V|)$, then the running time is reduced to $O(|V|b/\log b)$.

A further generalization is to consider a layered network $G = (V, E)$ with source s and terminal t, where each edge (i, j) has two associated nonnegative integer weights: a_{ij} is called the *distance* and $c_{i,j}$ is called the *time*. The problem is to find a s-t path of length at most b using at most d time. Exact and approximate algorithms for this problem have been studied in e.g. Lawler [14] and Hassin [10].

To solve the problem we introduce the binary table $f_{i,j}$ for $i \in V$, $j = 0, \ldots, b$. Here $f_{i,j} = 1$ iff a path of length j exists from s to i. A secondary table $g_{i,j}$ is 1 iff a path using at most time j exists from s to i.

Now, using the same approach as for the knapsack problem we merge the tables getting an algorithm with time complexity $O(|E|m/\log m)$ and space complexity $O(|V|m/\log m)$ where $m = \max\{b, d\}$. For values of b which are of magnitude $O(d)$ this improves the $O(|E|d)$ time complexity of the dynamic programming algorithm by Lawler [14].

5 Further generalizations

In this section we will apply the derived techniques to several optimization problems. The problems are described in the simple "subset-sum" version with straightforward generalisation to the "knapsack" version.

5.1 Bounded subset-sum and knapsack problem

The bounded subset-sum problem is the generalization of the ordinary subset-sum problem where each number may be chosen a bounded number of times. Thus given a set $S = \{(a_1, d_1), (a_2, d_2), \ldots, (a_n, d_n)\}$ of nonnegative integer pairs, and an additional integer b. The problem asks to maximize the sum $\sum_S a_i x_i$ without exceeding b, respecting that $x_i \in \{0, 1, \ldots, d_i\}$. The integers a_i are denoted the *weights*, d_i are denoted the *bounds* and b is denoted the *capacity*.

Using binary coding as proposed by Martello and Toth [15] we may convert the bounded subset-sum problem to an ordinary subset-sum problem with $\sum_{i \in S} \log d_i$ items. Since $d_i \leq b$ the number of items is $O(n \log b)$. Using the recursion from Section 2 we get a total running time of $O(nb)$ with space complexity $O(b/\log b)$. This improves the complexity of the Bellman recursion [2] which has time complexity $O(b \sum_{j=1}^{n} d_j)$ and space complexity $O(nb)$.

Using the same approach for the bounded knapsack problem we obtain an algorithm which runs in $O(nm)$ time and $O(m/\log m)$ space, where $m = \max\{b, z\}$.

5.2 Multiple-choice subset-sum and knapsack problem

In the multiple-choice subset-sum problem we have k classes S_h, $h = 1, \ldots, k$ of nonnegative integers $\{a_{h1}, a_{h2}, \ldots, a_{hn_h}\}$ and an additional integer b. The integers a_{hi} are denoted *weights* and b is the *capacity*. The problem asks to choose exactly one weight from each class such that the corresponding sum is maximized without exceeding b.

The problem may be formulated as a path problem in a layered network by letting the weights in class S_h correspond to nodes in layer $2h$ in the graph. Between layer $2h$ and $2h+2$ we introduce an intermediate node which is connected by an edge to all nodes at layer $2h$ and $2h+2$. The outgoing edges from an intermediate node $2h+1$ have distance equal to the integer values in S_h, and the ingoing edges from the intermediate node have an associated distance of zero.

The result from Section 4 immediately give a time complexity of $O(nb/\log b)$ where $n = \sum_{h=1}^{k} n_h$. The space complexity may however be reduced to $O(b/\log b)$ as we only need knowledge of three tables $f_{i,*}$ at any time: One for the result table in the intermediate node, one table for the previous intermediate node, and one table for the current node considered. This improves the complexity of $O(nb)$ time and $O(kb)$ space presented by [7].

The approach is easily generalized to the multiple-choice knapsack problem in which case we get the time complexity $O(nm/\log m)$ and space complexity $O(m/\log m)$ where $m = \max\{b, z\}$.

5.3 Unbounded subset-sum problem

Given a set $S = \{a_1, a_2, \ldots, a_n\}$ of nonnegative integers and an additional integer b, the unbounded subset-sum problem asks to maximize the sum $\sum_{i \in S} a_i x_i \leq b$ where $x_i \in \{0, 1, \ldots\}$. Although the unbounded subset-sum problem can be solved as the bounded subset-sum problem by imposing a trivial upper bound on the integers, a better dynamic programming recursion was presented by Gilmore and Gomory [8]. This algorithm reduces the time and space complexity to $O(nb)$ by allowing "self-references" in the recursion, i.e. the recursion at level i is allowed to use results obtained at the same level. Thus let $f_{i,j}$ for $i = 0, \ldots, n$, $j = 0, \ldots, b$ be a solution to the unbounded subset-sum problem defined on items $S_i = \{a_1, \ldots, a_i\}$ with $b = j$. Initially we set $f_{0,j} = 0$ for $j = 0, \ldots, b$. Subsequent values of f are derived as

$$f_{i,j} = \max\{f_{i-1,j}, f_{i,j-a_i} + a_i\} \tag{5}$$

In order to get a more compact representation based on word encoding, we let $g_{i,j} = 1$ if the sum $j \leq b$ can be obtained with items $1, \ldots, i$. This will ensure a space bound of $O(b/\log b)$.

As we may delete duplicated integers $a_i \in S$ at most $\log b$ of the integers will be smaller than $\log b$. Choose these integers first. Run a revised recursion, where in the i'th step the word-length is chosen equal to a_i. Thus if $a_i = 1$ we get an "ordinary" recursion of the form

$$g_{i,j} = 1 \quad \text{iff} \quad (g_{i-1,j} = 1 \text{ or } g_{i,j-a_i} = 1 \text{ or } g_{i-1,j-a_i} = 1) \tag{6}$$

If the above recursion is evaluated for increasing values of j this will ensure that the number a_i can be chosen an arbitrary number of times. If $a_i = 2$ we consider two bits in each iteration, and so on. The time complexity becomes less than

$$b + \frac{b}{2} + \frac{b}{3} + \frac{b}{4} + \cdots + \frac{b}{\log b} = b \sum_{i=1}^{\log b} \frac{1}{i} = bH_{\log b}$$

where the harmonic number H_n is of magnitude $\log n + O(1)$. This gives the total running time $O(b \log\log b)$ for the first part of the algorithm.

For the remaining integers with $a_i \geq \log b$ the following technique is used: Let g_{i-1} be a binary string representing those sums which can be obtained with the integers $a_1, \ldots, a_{i-1}$. Take the lowest order word in the binary string, shift it to the right by a_i positions and `or` it to g_{i-1}. Take the second-lowest order word in g_{i-1} and repeat the operation. Continue this way until all words in g_{i-1} have been considered. This implies that the value a_i can be added to the sum any number of times, since it is reintroduced to the sum. Still the complexity of a single merging of binary strings takes $O(b/\log b)$.

In total, the time complexity becomes $O(b \log\log b + nb/\log b)$. For $n > \log b \log\log b$ the latter term is however dominating. E.g. for a 32-bit word the last term is dominating for $n \geq 160$. Thus the time complexity is $O(nb/\log b)$ for sufficiently large values of n. This improves the complexity of the recursion by Gilmore and Gomory.

5.4 Change-making problem

Given a number of coin types $S = \{a_1, a_2, \ldots, a_n\}$ where a_i are nonnegative integers and an additional integer b. Choose any number of each coin such that the chosen coins sum to b and such that the minimum number of coins are used.

To solve this problem we first sort the coins $S = \{a_1, a_2, \ldots, a_n\}$ according to decreasing order of magnitude. Then we apply the algorithm for the unbounded subset-sum problem from Section 5.3. As soon as a value $g_{i,b} = 1$ is found, the algorithm terminates and outputs the combination of values leading to this solution. The algorithm will run in $O(n \log n) + O(nb/\log b)$ time and $O(b/\log b)$ space.

5.5 Two-partition

The two-partition problem may be formulated as follows: Given two sets of integers $S = \{a_1, a_2, \ldots, a_n\}$ and $T = \{a'_1, a'_2, \ldots, a'_{n'}\}$ and an integer b. Find two subsets $\overline{S}$ and $\overline{T}$ such that $\sum_{\overline{S}} a_i = \sum_{\overline{T}} a'_i$ is maximized without exceeding b. As for the subset-sum problem we run dynamic programming with word encoding for each set S and T. This gives us two binary strings $g_{n,*}$ and $h_{n,*}$. Now, use binary `and` to add these strings obtaining a new table f_*. In this string we have $h_j = 1$ iff it is possible to obtain $\sum_{\overline{S}} a_i = \sum_{\overline{T}} a'_i = j$, thus we may simply use linear search starting from b and running downward to find the largest value of j for which $h_j = 1$. The whole algorithm will run in $O((n + n')b/\log b)$ time and $O(b/\log b)$ space.

6 Computational experiments

We have restricted the computational experiments to the variant of subset-sum where the output of the algorithm should be a complete list of optimal solution values for all capacities up to b. Only the Bellman recursion [2], and the `wordsubsum` algorithm presented in this paper have the property of returning all subsolutions. Two versions of the Bellman recursion have been implemented. The `belltab` implementation is based on the straightforward recursion (2). The space complexity has however been reduced to $O(b)$ as we only need table $f_{i-1,*}$ to derive table $f_{i,*}$. The `bellstate` implementation is also based on the recursion (2) but only undominated entries where $f_{i,j} = j$ in (2) are saved, thus for "sparse" problems where very few subset sums can be achieved, this approach is more time and space effective. The basic algorithm for solving the recursion was taken from [20]. The space complexity of `bellstate` is also $O(b)$. None of the considered algorithms determine the subset $\overline{S}$ corresponding to an optimal solution value z, although this easily could be done by using the approach from Proposition 2.

To test the performance of the three algorithms, five types of data instances from Martello and Toth [15] were considered:

Table 1: Solution times in seconds, as average of 100 instances (AMD 1.1GHz). Algorithm `belltab` is in the upper part of the table, `bellstate` in the middle, and `wordsubsum` in the buttom of the table.

n	pthree	psix	evenodd	avis	todd
10	0.0000	0.4758	0.0001	0.0000	0.0111
30	0.0007	4.9177	0.0007	0.0010	□
100	0.0079	53.6850	0.0079	0.8922	□
300	0.1424	480.1904	0.1597	99.6214	□
1000	3.9393	—	3.9529	—	□
3000	35.8132	—	35.9105	□	□
10000	398.7499	□	399.9041	□	□
10	0.0000	0.0000	0.0000	0.0000	0.0000
30	0.0013	2.2375	0.0007	0.0002	□
100	0.0214	90.1765	0.0108	0.1072	□
300	0.7505	923.4170	0.1862	10.9937	□
1000	8.9221	—	4.3092	—	□
3000	80.0925	—	39.8775	□	□
10000	894.0435	□	448.7451	□	□
10	0.0000	0.0055	0.0000	0.0000	0.0001
30	0.0000	0.0966	0.0000	0.0001	□
100	0.0003	1.1294	0.0002	0.0043	□
300	0.0020	10.2859	0.0020	1.8627	□
1000	0.0218	117.3774	0.0214	235.1294	□
3000	0.2468	1061.2333	0.2450	□	□
10000	4.4961	□	4.9902	□	□

`pthree`	a_j uniformly random distributed in $[1, 10^3]$ and $b = \lfloor n10^3/4 \rfloor$
`psix`	a_j uniformly random distributed in $[1, 10^6]$ and $b = \lfloor n10^6/4 \rfloor$
`evenodd`	a_j even, uniformly random distributed in $[1, 10^3]$ and $b = 2\lfloor n10^3/8 \rfloor + 1$ (odd)
`avis`	$a_j = n(n+1) + j$ and $b = n(n+1)\ \lfloor (n-1)/2 \rfloor\ + n(n-1)/2$
`todd`	$a_j = 2^{\lfloor \log_2 n \rfloor + n + 1} + 2^{\lfloor \log_2 n \rfloor + j} + 1$ and $b = \lfloor \frac{1}{2} \sum_{j=1}^{n} a_j \rfloor$

Jeroslow [13] showed that every branch-and-bound algorithm enumerates an exponentially growing number of nodes when solving `evenodd` problems. Avis [3] showed that any recursive algorithm which does not use dominance will perform poorly for the `avis` problems. Finally Todd in [3] constructed the `todd` problems such that any algorithm which uses upper bounding tests, dominance relations, and rudimentary divisibility arguments will have to enumerate an exponential number of states.

The computational experiments were carried out on a AMD Thunderbird 1.1GHz with 768 Mb RAM. The word size of the processor is $w = 32$. The running times of the three algorithms are compared in Table 1, and a table of the obtained speedups is shown in Table 2. Entries marked with a □ could not be generated as the coefficients a_i do not fit within a 32-bit word, while entries marked with a dash could not be solved due to insufficient memory or time. As expected, the word encoding leads to a considerably more efficient algorithm which in general is at least one order of magnitude faster than the two other implementations. For medium sized problems of `pthree` and `evenodd` we even observe an

Table 2: Speedup by using `wordsubsum` in comparison to `belltab` (top) and `bellstate` (buttom).

n	`pthree`	`psix`	`evenodd`	`avis`	`todd`
100	26	48	40	207	—
300	71	47	80	53	—
1000	181	—	185	—	—
3000	145	—	147	—	—
10000	89	—	80	—	—
100	71	80	54	25	—
300	375	90	93	6	—
1000	409	—	201	—	—
3000	325	—	163	—	—
10000	199	—	90	—	—

improvement of 200–400 times. This may seem surprising as the word size is $w = 32$, but due to the decreased space complexity caching gets improved, which results in an additional speedup. For large problems, which do not fit within the cache, the speedup decreases.

It is also interesting to observe the fundamental difference between the two data structures used for the Bellman recursion. For "sparse" problems like the `avis` and `todd` instances it is more efficient to use the state representation, while the table representation is more efficient for problems where nearly all the possible weight sums can be derived. The `wordsubsum` algorithm is based on a table representation, and thus it performs best for the "dense" problems. But even for the "sparse" problems it is able to beat the state representation by almost an order of magnitude, and `wordsubsum` is able to solve larger problems due to the reduced space complexity.

Finally Table 3 compares the solution times of state-of-art algorithms for the subset-sum problem where only the solution for capacity b is demanded. The first algorithm is the `decomp` algorithm [20] which is a dynamic programming algorithm based on Horowitz-Sahni decomposition of the problem. The next algorithm is the `mtsl` algorithm by Martello and Toth [15] which is a branch-and-bound algorithm. The following `balsub` algorithm is based on balanced dynamic programming [19], making use of the fact that every optimal solution will have a total weight sum very close to b and thus only weight sums in an interval around b need to be considered. Finally the solution times of `wordsubsum` are repeated for completeness of the table. As seen from the table, `wordsubsum` has similar solution times as the other algorithms for the difficult instances `evenodd`, `avis` and `todd`, although `wordsubsum` solves the problem for all values of the capacity up to b. For the easy instances, where a huge amount of optimal solutions exists, the performance of `wordsubsum` is not as good. On the other hand `wordsubsum` solves the problem for all different values of the capacity up to b faster than solving each individual capacity by e.g. `decomp`.

Table 3: Solution times in seconds, as average of 100 instances (AMD 1.1GHz). Algorithm `decomp` (first), `mtsl` (second), `balsub` (third) and `wordsubsum` (buttom).

n	`pthree`	`psix`	`evenodd`	`avis`	`todd`
10	0.0000	0.0000	0.0000	0.0000	0.0000
30	0.0000	0.0003	0.0007	0.0001	□
100	0.0000	0.0003	0.0157	0.0216	□
300	0.0001	0.0001	0.3404	2.8849	□
1000	0.0000	0.0001	4.2096	420.5549	□
3000	0.0000	0.0002	37.5873	□	□
10000	0.0002	□	414.7411	□	□
10	0.0001	0.0000	0.0000	0.0001	0.0000
30	0.0000	0.0007	0.4299	1.1669	□
100	0.0000	0.0007	—	—	□
300	0.0000	0.0008	—	—	□
1000	0.0001	0.0008	—	—	□
3000	0.0001	0.0008	—	□	□
10000	0.0000	□	—	□	□
10	0.0001	0.7015	0.0001	0.0001	0.0159
30	0.0001	1.2281	0.0004	0.0004	□
100	0.0001	0.6302	0.0011	0.0164	□
300	0.0001	0.4145	0.0032	1.9966	□
1000	0.0000	0.3380	0.0100	73.9668	□
3000	0.0000	0.3832	0.0301	□	□
10000	0.0000	□	0.0979	□	□
10	0.0000	0.0055	0.0000	0.0000	0.0001
30	0.0000	0.0966	0.0000	0.0001	□
100	0.0003	1.1294	0.0002	0.0043	□
300	0.0020	10.2859	0.0020	1.8627	□
1000	0.0218	117.3774	0.0214	235.1294	□
3000	0.2468	1061.2333	0.2450	□	□
10000	4.4961	□	4.9902	□	□

7 Conclusion

The present paper has introduced a number of general techniques which may be used to decrease time and space complexity of dynamic programming algorithms by use of word encoding: Section 2 showed the basic principle of encoding and Proposition 2 demonstrated how the optimal solution corresponding to a given solution value can be found without any auxiliary information. Section 3 dealt with the case where the objective function and constraint differ, while Section 5.3 showed how "self-references" in dynamic programming recursions can be handled by emulating increasing word sizes. Thus a quite comprehensive toolbox has been developed making it easier to apply the framework in future dynamic programming recursions. The numerous examples in Section 5 demonstrate that the toolbox can be applied to a variety of problems. As the recursions are general they can be combined with other techniques for reducing the computational times. This could e.g. be the concept of solving a minimal core problem for the knapsack problems [18]. As several fully polynomial approximation schemes are based on dynamic programming with some scaled coefficients, the word encoding may lead to improved running times for these schemes. It is however not trivial to apply the word encoding approach for approximations schemes as this would demand that the coefficients in both the objective function and the constraints are scaled to achieve a reduced complexity. Most approximation schemes only scale one of the coefficient sets.

Throughout the paper we have assumed that $w = \Theta(\log m)$ i.e. that the word size is of the same magnitude as the coefficients in the input and output data. In practice one will normally use the largest possible word size supported by the processor, thus obtaining an additional decrease in computational time. Modern processors like the Pentium IV processor support 128 bit integer arithmetics [12], and thus word parallelism may be of significant importance. As dynamic programming algorithms in general are very space consuming, word encoding makes it possible to solve larger problems. Reducing the space complexity also improves caching of the processor, thus leading to an additional decrease in the observed computational time. Algorithms based on word encoding also avoid conditional statements in the code, thus being well suited for modern pipelined processors.

From a theoretical point of view the reduced time and space complexity in dynamic programming is an interesting result. In addition, it has been demonstrated that the dynamic programming recursions for the subset-sum like problems also in practice leads to considerably faster algorithms. For the knapsack like problems the presented algorithms have quite large constants hidden in the time complexity terms, and thus in practice they will hardly compete with existing algorithms. The reduced space complexity may however be very attractive as it improves caching and makes it possible to solve larger problems. Combining a classic dynamic programming recursion with the reduced space complexity — converting between the two representations on-line — could be a direction to investigate. An alternative direction to go could be to implement the action table from Proposition 4 in hardware in future processors.

As a concluding remark we will quote Hagerup [9]: "it is not clear at the time of writing whether word-based algorithms will exert a direct practical influence". The present paper has demonstrated that word-based techniques, when applied to the proper problems, indeed are of practical relevance.

References

[1] Andersen Consulting, Karigma Project (2000).

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

[3] V. Chvátal (1980), "Hard Knapsack Problems", *Operations Research*, **28**, 1402–1411.

[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest (1989), "Introduction to Algorithms", MIT-Press, Cambridge, England.

[5] G. B. Dantzig (1957), "Discrete Variable Extremum Problems", *Operations Research*, **5**, 266–277.

[6] M. Desrochers, J. Desrosiers and M. Solomon (1992), "A New Optimization Algorithm for the Vehicle Routing Problem with Time Windows" *Operations Research*, **40**, 342–354.

[7] K. Dudziński and S. Walukiewicz (1987), "Exact Methods for the Knapsack Problem and its Generalizations", *European Journal of Operational Research*, **28**, 3–21.

[8] P.C. Gilmore and R.E. Gomory (1965), "Multi-stage cutting stock problems of two and more dimensions", *Operations Research*, **13**, 94–120.

[9] T. Hagerup (1998), "Sorting and searching on the word RAM" Proc. 15th Symposium on Theoretical Aspects of Computer Science (STACS 1998), *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, **1373** 366–398.

[10] R. Hassin (1992), "Approximation schemes for the restricted shortest path problem", *Mathematics of Operations Research*, **17**, 36–42.

[11] T. Ibaraki (1987), "Enumerative Approaches to Combinatorial Optimization – Part 2", *Annals of Operations Research*, **11**, 376–388.

[12] Intel Pentium 4 Processor Optimization, Reference Manual, 1999-2000, Intel Corp (`http://developer.intel.com`).

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

[14] E.L. Lawler (1976), *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York.

[15] S. Martello and P. Toth (1990), *Knapsack Problems: Algorithms and Computer Implementations,* Wiley, Chichester, England.

[16] G. L. Nemhauser and Z. Ullmann (1969), "Discrete dynamic programming and capital allocation", *Management Science*, **15**, 494–505.

[17] U. Pferschy (1999) "Dynamic programming revisited: Improving knapsack algorithms", *Computing*, **63**, 419–430.

[18] D. Pisinger (1997), "A minimal algorithm for the 0-1 knapsack problem", *Operations Research*, **45**, 758–767.

[19] D. Pisinger (1999), "Linear time algorithms for knapsack problems with bounded weights", *Journal of Algorithms*, **33**, 1-14.

[20] D. Pisinger (1999), "An exact algorithm for large multiple knapsack problems", *European Journal of Operational Research*, **114** 528–541.

[21] D. Pisinger and P. Toth (1998), "Knapsack Problems". In: D-Z. Du, P. Pardalos (ed.), *Handbook of Combinatorial Optimization, Vol 1*, Kluwer Academic Publishers 299–428.

Appendix

The `wordsubsum` algorithm can be described by the following code written in C. The arguments to `wordsubsum` are: `n` which is the number of weights, `b` which is the capacity, `a[]` which is the array $\{a_1, \ldots, a_n\}$ of weights, and `g[]` which is a binary table where the solutions should be returned. Notice that `g[]` must have length at least `L+1` where $\texttt{L} = \lceil \texttt{b}/\texttt{W} \rceil$ and `W` is the word size.

```
typedef unsigned int word;    /* unsigned integer              */
const word W = 32;            /* bits in an integer            */
const word B = 5;             /* logarithm to W                */

void wordsumbsum(int n, int b, int a[], word g[])
{
  int h, i, j, L, S;
  word bs1, bs2, ws;

  /* initialize g */
  g[0] = 1;
  for (j = 1; j < L; j++) g[j] = 0;

  /* now run the recursion for items i = 1,..,n */
  S = 0;                        /* sum of weights                */
  L = (b >> B);                 /* index of integer holding bit b */
  for (h = 0; h < n; h++) {     /* for all weights 1,...,n       */
    ws = (a[h] >> B);           /* number of words to shift      */
    bs1 = a[h] - (ws << B);     /* number of bits to shift       */
    bs2 = W - bs1;              /* remaining number of bits      */
    S += a[h];                  /* S = a[1] + .. + a[h]          */
    i = MINIMUM(L, S >> B);     /* last position in g[] to read  */
    j = i - words;              /* last position in g[] to write */
    if (bs1 != 0) {
      for (; j > 0; j--, i--) g[i] |= ((g[j] << bs1) | (g[j-1] >> bs2));
      g[i] |= (g[j] << bs1);
    } else {
      for (; j >= 0; j--, i--) g[i] |= g[j];
    }
  }
}
```