A Parallel Algorithm for the Knapsack Problem

EHUD D. KARNIN, MEMBER, IEEE

Abstract — **A time–memory-processor tradeoff for the knapsack problem is proposed. While an exhaustive search over all possible solutions of an n-component knapsack requires $T = O(2^n)$ running time, our parallel algorithm solves the problem in $O(2^{n/2})$ operations and requires only $O(2^{n/6})$ processors and memory cells. It is an improvement over previous time–memory-processor tradeoffs, being the only one which outperforms the $C_mC_s = 2^n$ curve. C_m is the cost of the machine, i.e., the number of its processors and memory cells, and C_s is the cost per solution, which is the product of the machine cost by the running time.**

Index Terms — **Cryptography, knapsack problem, parallel architecture, time–memory-processor tradeoff, VLSI complexity.**

I. INTRODUCTION

GIVEN n positive integers $(a_1, a_2, \cdots, a_n)$ and a positive integer S, the knapsack problem is to find a binary solution $(x_1, x_2, \cdots, x_n)$, $x_i \in \{0, 1\}$ for the equation

$$\sum_{i=1}^{n} a_i x_i = S. \tag{1}$$

The associated decision problem, i.e., determining whether a solution exists, is in the class of NP-complete problems [4]. Hence, (1) is considered to be a very hard problem, making it a good basis for various cryptographic schemes [3].

An n component knapsack has $N = 2^n$ possible solutions to search over, a task which can be accomplished in N trials if *exhaustive search* is used. However, even for $n = 50$ and assuming 1 μs per trial, it will take more than 35 years to enumerate all the solutions. Thus, it may be desirable to reduce the time T at the expense of utilizing more memory M. For example, a simple *time–memory tradeoff* algorithm [5], to be described in the next section, solves an n component knapsack in time $T = 2^{(1-\alpha)n}$, if $M = 2^{\alpha n}$ words of memory are available, for $0 \leq \alpha \leq 1/2$. For $n = 50$ and a storage of about 1 million words (2^{20}), the problem is solved in only 18 min, assuming a 1 μs per operation machine.

Traditionally, time and memory were the only resources considered when the computational complexity of an algorithm had to be evaluated. (Memory is also known as *space*, especially when it is referred to in complexity theory.) The architecture of the computing machine was a single processor sequentially addressing the memory cells. This is a reasonable model as long as processors are much more expensive than storage space.

Manuscript received August 8, 1983. This work was supported in part by National Science Foundation Grant ECS-79-16161, JSEP Grant DAAG29-81-0057, and NSA Grant MDA904-81-C-0414. This paper was presented at the IEEE International Symposium on Information Theory, St. Jovite, Quebec, Canada, September 26–30, 1983.

The author was with the IBM Research Laboratory, San Jose, CA 95193. He is now with the IBM Scientific Center, Technion City, Haifa 32000, Israel.

Recent developments in VLSI substantially reduced the processor-to-memory cost ratio. In this technology, the cost of each feature is proportional to the area it consumes on the silicon wafer, and processors and memory cells have areas of comparable size. As an example, a 1 bit Adder is only about 10 times larger than a 1 bit memory cell. (We shall later see that addition, or subtraction, is the only operation the processors have to perform in order to execute our new algorithm.) The new developments suggest looking at novel architectures, such as in [2], which make use of parallel computation as a means of achieving better cost performance in solving large problems. The tradeoff between time and memory is extended to a *time–hardware tradeoff*, where the hardware H is a combination of processors and memory. To be precise,

$$H = c_pP + c_mM \tag{2}$$

where P and M are the numbers of processors and memory cells, respectively, c_p is the cost of a single processor, and c_m is the cost of one word of memory. Throughout this paper, we neglect constant and even logarithmic factors in N (i.e., factors which are polynomial in n); therefore, (2) is simplified to

$$H = \max(p, M). \tag{3}$$

Another justification for introducing parallelism has an even deeper reason than technological innovations. Currently, the most efficient sequential (i.e., single processor) algorithm for solving general knapsack problems needs time $T = 2^{n/2}$, with memory $M = 2^{n/4}$ [1], [7]. Time and memory may be traded off according to

$$M^2T = N \tag{4}$$

but this relation holds only for $T \geq 2^{n/2}$. Suppose we want to use this procedure for solving a 100 component knapsack. While 2^{25} word of memory are within reach, 2^{50} operations take more than 35 years even on a fast 1 μs per operation computer. Dividing the searching task among 1000 processors, as suggested in [7], reduces the time to less than two weeks, a realistic period for finding a solution.

We notice that the remedy which shortened the lengthy computation time, required by the sequential algorithm, was the introduction of parallelism. Clearly, every search problem is amenable to a simple time–hardware tradeoff of the form $HT = N$. Simply partition the N points of the search space into H equal subsets, and assign a processor to search over the $T = N/H$ points of each subset. The previous example demonstrated that even this trivial tradeoff is sometimes necessary and useful. So, if parallelism is essential in

0018-9340/84/0500-0404$01.00

overcoming some fundamental limitations of sequential algorithms, it is worthwhile to explore better ways of exploiting a multiprocessor system.

Parallel algorithms for various problems have been devised in recent years. They are often analyzed in the context of VLSI performance evaluation, and present *time–area tradeoffs* for certain computational tasks. Amirazizi and Hellman [2] investigated the implications of parallelism to cryptanalysis. They found a general tradeoff between C_m, the cost per machine, and C_s, the cost per solution (of a search problem), which obeys

$$C_m \cdot C_s = N \tag{5}$$

where N is the number of points in the search space. The cost of the machine is merely the cost of its hardware H. The cost of running a program on this machine is proportional to both H and the run time T, so $C_s = HT$, and (5) may be rewritten as

$$H^2T = N. \tag{6}$$

Fig. 1 shows C_s versus C_m on a logarithmic scale for various algorithms that solve the knapsack problem. Point 1 corresponds to an exhaustive search where $H = 1$ and $T = 2^n$. (Recall that constant and logarithmic factors are neglected.) Point 2 is the *two list algorithm* [5], described in the next section, which uses only one processor. (With $2^{n/2}$ processors, the same point is achieved by the trivial $HT = N$ time–hardware tradeoff mentioned above.) The straight line in Fig. 1 corresponds to (5). Its solid part can be obtained while solving a single problem, whereas the dashed part may be achieved only by dividing the cost of running the program, which equals HT, by an exponential (in n) number of problems solved simultaneously [2]. (To formalize, the cost per solution should be computed as $C_s = HT/S$, where S stands for the simultaneity of the solution.) While the segment between points 1 and 3 can be obtained by the multiprocessor system presented in [2], it can also be achieved by a single processor using the *four table algorithm* [1], [7].

Point 5 corresponds to the new algorithm developed in Section III. It solves an n component knapsack problem in time $T = 2^{n/2}$ with only $H = 2^{n/6}$ processor and memory cells. As a time–hardware tradeoff, it may be extended to the straight line connecting points 5 and 1 in Fig. 1, which means that it obeys the equation

$$H^3T = N \tag{7}$$

for $T \geq 2^{n/2}$. Currently, this is the only method we know that outperforms the $C_mC_s = N$ tradeoff curve.

II. The Basic Two List Algorithm

Given an n component knapsack $a = (a_1, a_2, \cdots, a_n)$ and a sum S, we now describe a simple algorithm, which solves the problem in time $T = 2^{n/2}$ with $M = 2^{n/2}$ words of memory. This method is reported in [5] and is termed as a "two list algorithm" for a reason which will shortly become obvious.

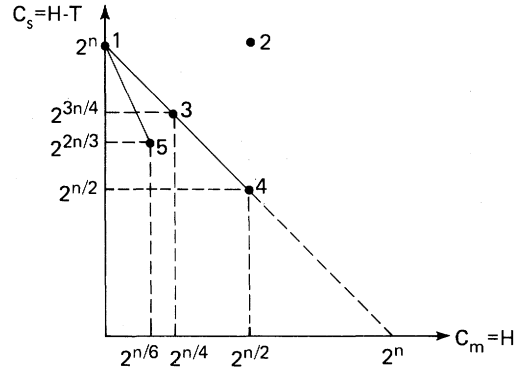


Fig. 1. Time–hardware tradeoffs for the knapsack problem.

A. Two List Algorithm

Precomputation:

1) Divide a into two equal parts, $(a_1, \cdots, a_{n/2})$ and $(a_{n/2+1}, \cdots, a_n)$.

2) Form all $2^{n/2}$ possible subset sums of $(a_1, \cdots, a_{n/2})$, sort them in an *increasing order* and store as list $A = (A_1, A_2, \cdots, A_{2^{n/2}})$.

3) Form all $2^{n/2}$ possible subset sums of $(a_{n/2+1}, \cdots, a_n)$, sort them in a *decreasing order* and store as list $B = (B_1, B_2, \cdots, B_{2^{n/2}})$.

Computation:

1) (Initialize) $i \leftarrow 1, j \leftarrow 1$.

2) If $A_i + B_j = S$, stop; a solution is found.

3) If $A_i + B_j < S$, then $i \leftarrow i + 1$, else $j \leftarrow j + 1$.

4) If $i > 2^{n/2}$ or $j > 2^{n/2}$, stop; there is no solution.

5) Go to step 2).

Remark: This algorithm is capable of solving any knapsack problem, not necessarily a one-to-one knapsack, so it may happen that some elements in a list are identical. Therefore, the terms "increasing" and "decreasing" should be interpreted in the broad sense as nondecreasing and nonincreasing, respectively. The same remark applies to the other two algorithms, to be described later.

A small example might be helpful here. Consider the knapsack vector (24, 57, 13, 75, 5, 30) with $S = 59$. It is broken into (24, 57, 13) and (75, 5, 30), and the two sorted lists of sub set sums are

A	B
0	110
13	105
24	80
37	75
57	35
70	30
81	5
94	0

The computation proceeds by checking that $0 + 110 > 59$, $0 + 105 > 59$, etc., until it finds that $A_3 + B_5 = S$, i.e., $24 + 35 = 59$, and then stops.

It is not hard to see that this algorithm actually finds a solution, if it exists. Any positive integer which solves the problem can be represented as a sum of the form $A_i + B_j$ for some i, j. Because B_1 is the largest element in the B-list, if

$A_1 + B_1 < S$ then any other sum which involves A_1 is also too small, and A_1 can be deleted from the A-list. The new A-list, which starts with A_2, exhibits the same behavior; thus, our claim is inductively verified. Similarly, if $A_1 + B_1 > S$, then B_1 can be deleted, obtaining a new list, starting with B_2, which is in a decreasing order. Thus, for this case too, our claim is proved by induction.

The performance of the algorithm is readily obtained. Memory is required to store two lists with $2^{n/2}$ terms, so $M = O(2^{n/2})$. The computation progresses as long as $A_i + B_j \neq S$, but in each step either i or j is incremented by 1. At worst, it terminates when both lists are exhausted, hence $T = O(2^{n/2})$. Moreover, the precomputation can also be completed in $2^{n/2}$ operations since sorting $\sqrt{N}$ elements can be done in $O(\sqrt{N} \log \sqrt{N})$ operations [6] (and we neglect logarithmic factors).

The two list algorithm offers a tremendous improvement over an exhaustive search. It also serves as the starting point for introducing a more cost effective parallel method. The basic algorithm carries potential improvements in two dimensions: time and hardware. We were unable to reduce the computation time T from $2^{n/2}$, except by the trivial method of allocating processors to subsets of the search space, as mentioned before. Thus, the problem posed by Schroeppel and Shamir [7], whether $2^{n/2}$ is a lower bound for T, is still open. However, we did improve on the hardware by reducing it from $2^{n/2}$ to $2^{n/6}$.

III. Description of the Parallel Algorithm

We observe that the basic algorithm is executed by traversing through ordered lists. Since each list is passed in one direction only, i.e., no backtracking is required, there might be a fast way to generate the terms of the list "on-line," rather than storing all of them. Indeed, such a method exists [1], [7] and is known as the "four table algorithm." We describe how to generate the elements of the increasing list A. The generation of the decreasing list B is similar.

A. Four Table Algorithm

Precomputation:

1) Divide $(a_1, \cdots, a_{n/2})$ into two unequal parts, $(a_1, \cdots, a_{n/3})$ and $(a_{n/3+1}, \cdots, a_{n/2})$.

2) Form all subset sums of $(a_1, \cdots, a_{n/3})$, sort them in an increasing order, and store as list $P = (p_1, p_2, \cdots, p_{2^{n/3}})$.

3) Form all $2^{n/6}$ possible subset sums of $(a_{n/3+1}, \cdots, a_{n/2})$ and denote them by $Q = (q_1, q_2, \cdots, q_{2^{n/6}})$.

4) Build a priority queue with the elements $\{(p_1, q_i)\}_{i=1}^{2^{n/6}}$, which are ordered according to the *sums* of the pairs of values $\{p_1 + q_i\}$.

Note: Unlike the previous two list algorithm where the first element of the increasing list A was *deleted* if not used, here we shall need to repeatedly *replace* the smallest element. The new value is not necessarily the smallest number in the new queue, and hence needs to be *inserted* in its proper place in the table. This calls for implementing the priority queue as a heap [6], i.e., a binary tree structure. The root stores the smallest element in the queue, and each internal node contains a smaller number than its two descendants. For an m element heap, replacing the top element and reordering, takes $O(\log m)$ operations. One just checks whether the value that occupies an internal node is larger than either of its descendants, and if so, exchanges it with the smaller one. This process may proceed; that is, the new value propagates down the heap for at most $\log_2 m$ steps which is the depth of the binary tree.

The Repeated Steps:

1) Replace the current pair (p_j, q_k) at the top of the heap by (p_{j+1}, q_k).

2) Reorder the priority queue (according to the sums $\{p + q\}$).

3) Read the first element in the queue as the next A_i.

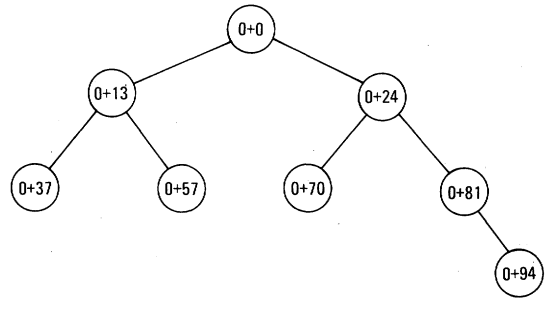
We clarify the description of this algorithm by an example. Let $(a_1, \cdots, a_{n/2}) = (50, 60, 70, 80, 90, 100, 24, 57, 13)$, which is broken into $(50, 60, 70, 80, 90, 100)$ and $(24, 57, 13)$. The P-list is

$$0, 50, 60, 70, \cdots, 450,$$

while the q elements are

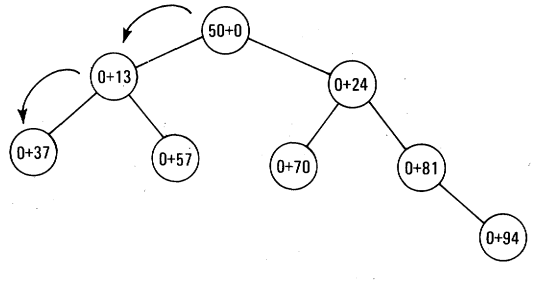
$$0, 13, 24, 37, 57, 70, 81, 94.$$

Hence, a possible initial state for the priority queue is



with $A_1 = 0$ at the top.

Now $(p_1, q_1) = (0, 0)$ at the top is replaced by $(p_2, q_1) = (50, 0)$. Reordering is necessary and is accomplished by two interchanges, as indicated by the arrows in the following illustration.



The new arrangement of the queue has $(0, 13)$ at the top. Notice that the next time an element from the P-list is recalled, it is p_2 again.

When the precomputation is terminated $A_2 = 13$ resides at the top of the heap. By the end of the reordering in the repeated steps, the next A_i is created because each A_i is composed as a sum $p_j + q_k$, and these sums are generated in an

increasing order. Hence, we have just shown how to generate the A_i's in a *logarithmic time*. The memory requirements are $O(2^{n/6})$ words for the priority queue, and $O(2^{n/3})$ for the ordered list P. The next step would be to eliminate the long list P, thus keeping the hardware requirements at $2^{n/6}$.

Remark: If only a single processor is available, the vector $(a_1, \cdots, a_{n/2})$, at the first step of the precomputation should be divided into two parts of equal size, $(a_1, \cdots, a_{n/4})$ and $(a_{n/4+1}, \cdots, a_{n/2})$. The hardware requirements will be $2^{n/4}$ words of memory for both the priority queue and the list P. Hence, an $M = 2^{n/4}$ and $T = 2^{n/2}$ algorithm for solving knapsacks results [1], [7].

Our next observation is that storing the list P would not be necessary, if there is a fast way to generate its elements. Unlike the former discussion where the A_i was always retrieved in an increasing order, the repeated step in the four table algorithm may recall the p_i in *any* order (i.e., backtracking does occur). Because (p_j, q_k) is replaced by (p_{j+1}, q_k), we need a machine which accepts an arbitrary p_j at its input and outputs p_{j+1} after a constant or logarithmic (in N) delay. Such a multiprocessor algorithm is described below.

Our machine is composed of $2 \cdot 2^{n/6}$ processors, and the only arithmetic operation they need to perform is subtraction. Also, each processor has to store one number which is loaded during the precomputation.

Precomputation:

1) Break $(a_1, \cdots, a_{n/3})$ into two parts, $(a_1, \cdots, a_{n/6})$ and $(a_{n/6+1}, \cdots, a_{n/3})$.

2) Compute all the subset sums of $(a_1, \cdots, a_{n/6})$ and sort them in an increasing order, denoting the resulting numbers as $(\alpha_1, \alpha_2, \cdots, \alpha_{2^{n/6}})$.

3) Compute all the subset sums of $(a_{n/6+1}, \cdots, a_{n/3})$ and sort them in an increasing order, denoting the resulting numbers as $(\beta_1, \beta_2, \cdots, \beta_{2^{n/6}})$.

Let t be the current value of p_j from the element (p_j, q_k) at the top of the heap; then our objective is to generate its successor in list P. In other words, we are interested in

$$\min_i \{p_i \mid p_i > t\}. \tag{8}$$

Note that a strict inequality is used in (8). Even in a many to one knapsack, it is not necessary to consider a p which is equal to the current value of t because this value failed to provide a solution to the problem at hand.

Since every p_i can be represented as a sum of the form $\alpha_j + \beta_k$ for some j and k, (8) may be rewritten as

$$\min_{j,k} \{\alpha_j + \beta_k \mid \alpha_j + \beta_k > t\}. \tag{9}$$

Searching for the minimum in (9) is equivalent to looking for

$$\min_{j,k} \{\alpha_j - (t - \beta_k) \mid \alpha_j - (t - \beta_k) > 0\}. \tag{10}$$

Once the smallest difference in (10) is found, it is added to the known value of t to obtain the next term of list P [see (8)]. The computation of expression (10) is accomplished by performing the following steps.

Computation:

1) Each β-processor computes $t - \beta_k$.

2) The processors *merge-sort* the two lists $(\alpha_1, \cdots, \alpha_{2^{n/6}})$ and $(t - \beta_1, \cdots, t - \beta_{2^{n/6}})$ into one ordered (increasing) list. In case of equal α and $t - \beta$ values, the α-value should precede the $(t - \beta)$-value. The combined list is stored within the linear array of $2 \cdot 2^{n/6}$ processors, and each processor also remembers the origin of its contents (whether it came from the α-list or the $(t - \beta)$-list).

3) Let PR_i denote the value stored at the ith processor with the convention that $PR_0 = 0$, and it originated from the $(t - \beta)$-list. Each processor does the following computation to determine if it has an α-value such that $\alpha - (t - \beta) > 0$. Note that this computation depends *only* on the processor's contents and that of its immediate predecessor. If PR_i has an α-origin, *and* PR_{i-1} has a $(t - \beta)$-origin, *and* $PR_i - PR_{i-1} > 0$, then

$$PR_i \leftarrow PR_i - PR_{i-1}.$$

Else $PR_i \leftarrow \infty$ (i.e., this is not a possible $\alpha - (t - \beta)$ value).

4) Compute $\min_i PR_i$, using a comparison tree. The leaves of the binary tree are the $2 \cdot 2^{n/6}$ processors, and each internal node is a comparator that finds the smaller of its two inputs and forwards it to its adjacent node towards the root of the tree.

Prior to analyzing the performance achieved by using this procedure, we work out a small example for illustrative purposes. Suppose that the α and β lists are

$$(\alpha_1, \cdots, \alpha_1) = (1, 5, 7, 8)$$
$$(\beta_1, \cdots, \beta_4) = (2, 4, 5, 6)$$

and $t = 5$.

Remark: The algorithm works on *any* two lists, not just 2^m subset sums of an m-component vector. The values in this example are not subset sums of any (partial) knapsack and were chosen because they illustrate more situations than a real knapsack of the same size.

After step 1), the $(t - \beta)$-list is modified to

$$(t - \beta_1, \cdots, t - \beta_4) = (3, 1, 0, -1)$$

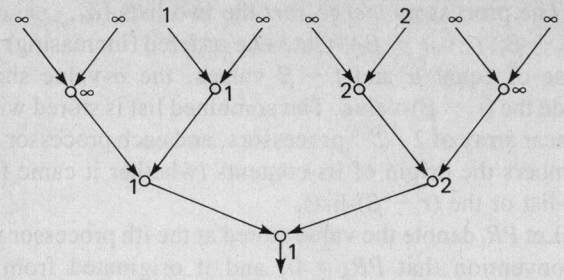
(which is also a sorted list, although the order is reversed). The two sorted lists are merged and sorted into one list in step 2), as

$$(PR_1, \cdots, PR_8) = (-1[\beta], 0[\beta], 1[\alpha], 1[\beta], 3[\beta], 5[\alpha], 7[\alpha], 8[\alpha])$$

where the origin of each PR_i is indicated in brackets. Now each processor computes the difference between the value it holds to that of its immediate left neighbor. It keeps the result if the three conditions of step 3) above are met, or else changes it to ∞. This process yields the following values:

$$(PR_1, \cdots, PR_8) = (\infty, \infty, 1, \infty, \infty, 2, \infty, \infty).$$

The last step, number 4), calls for a comparison tree to find the minimum of the numbers above. This is implemented, as shown below.



The smallest value, which equals 1 in this example, appears at the root of the tree. Now, the next p can be found as $t + 1 = 6$.

To verify the correctness of the parallel algorithm, we first recall the discussion previous to the derivation of expression (10). As explained there, it suffices to find the minimum in (10) in order to obtain the next element on the P-list, so we have to show that the four steps of the computation do find the minimum. Specifically, the only thing that needs verification is that evaluating the $2 \cdot 2^{n/6}$ differences in step 3) is enough, and one does not have to consider all $O(2^{n/3})$ possible pairs $(\alpha_j, t - \beta_k)$ in searching for the minimum. This is guaranteed by the increasing order of the merged list. If PR_i contains an α value, then a subtraction of PR_j, for $j > i$, yields a nonpositive result, hence should not be done. Also, for $j = 0, 1, \cdots, i - 2$, evaluating $PR_i - PR_j$ is wasteful because PR_{i-1} is greater than all its predecessors on the list, hence $PR_i - PR_{i-1} \leq PR_i - PR_j, j = 0, 1, \cdots, i - 2$.

Convinced that the algorithm is correct, we examine its performance in terms of hardware and computation time. Our device contains $2 \cdot 2^{n/6}$ processors, each one requiring only a constant storage space. The binary comparison tree can be implemented by putting a processor of the same type at each internal node of the tree. (Comparing two numbers is done by subtraction and checking the sign-bit of the result.) The tree has $2 \cdot 2^{n/6}$ leaves, hence it requires $2 \cdot 2^{n/6} - 1$ comparators, keeping the total amount of hardware at $H = O(2^{n/6})$.

Proceeding to timing analysis, the first step is clearly done in one unit of time because each processor acts individually. The second step, merging two *sorted* lists into one *sorted* list, is a well-studied problem. Batcher's algorithm [6] accomplishes the merging of two lists of size m in $\log_2 m$ steps, when $2m$ processors are used. Therefore, step 2) lasts for only $n/6$ units of time. The subtractions of step 3) are accomplished in one time unit, while it takes $n/6$ steps to propagate through the depth of the comparison tree. Summarizing, the computation terminates in $O(n)$ time, and because such logarithmic (in N) factors are neglected in our analysis, $T = 1$.

IV. Summary

The last section described a system that is capable of finding the smallest sum over all two element combinations, the elements taken from two given lists, where the sum has to exceed a prescribed value. For lists of length $2^{n/6}$, this can be done in one unit of time (neglecting logarithmic factors) with $O(2^{n/6})$ processors.

Combining this parallel system with the techniques of the two list and the four table algorithms, we obtain a method for solving a general n component knapsack problem in time $T = 2^{n/2}$ with $H = 2^{n/6}$ hardware. The hardware requirements are $O(2^{n/6})$ memory words as a working space for handling the priority queue, and $O(2^{n/6})$ processors, each possessing a constant storage space, for subtracting, merging, and forming the comparison tree.

Initial values for the processors and the priority queue are derived during the precomputation steps. A single processor can compute the required $O(2^{n/6})$ numbers and sort them in $T = 2^{n/6}$, which is negligible compared to the $O(2^{n/2})$ operations taken through the computation steps. Using all $O(2^{n/6})$ processors and their interconnecting Batcher's network enables doing even better, that is, to finish the precomputation in $T = 1$.

We conclude by referring again to Fig. 1. $H = 2^{n/6}$ and $T = 2^{n/2}$ corresponds to point 5 there. Obtaining the time–memory processor tradeoff of (7), which is the straight line from point 5 to 1, is easy. Apply our algorithm to a knapsack of $(1 - \varepsilon)n$ elements, searching exhaustively over the solution to the last εn terms. Each of the $2^{\varepsilon n}$ problems of length $(1 - \varepsilon)n$ is solved in $T = 2^{(1-\varepsilon)n/2}$ and $H = 2^{(1-\varepsilon)n/6}$; thus, going through all of them increases T to $2^{(1+\varepsilon)n/2}$. We check that

$$H^3T = [2^{(1-\varepsilon)n/6}]^3 \cdot 2^{(1+\varepsilon)n/2} = 2^n, \qquad 0 \leq \varepsilon \leq 1,$$

in accordance with (7). We do not know, however, whether it is possible to extend (7) below point 5 in Fig. 1, that is, to continue the *straight* line in that direction.

Acknowledgment

The author wishes to thank Prof. M. E. Hellman of Stanford University for very helpful discussions.

References

[1] J. H. Ahrens and G. Finke, "Merging and sorting applied to the zero-one knapsack problem," *Oper. Res.*, vol. 23, pp. 1099–1109, 1975.
[2] H. Amirazizi and M. Hellman, "Time-memory-processor trade-offs," *IEEE Trans. Inform. Theory,* submitted for publication.
[3] W. Diffie and M. E. Hellman, "Privacy and authentication: An introduction to cryptography," *Proc. IEEE,* vol. 67, pp. 397–427, Mar. 1979.
[4] M. Garey and D. Johnson, *Computers and Intractability—A Guide to the Theory of NP-Completeness.* San Francisco, CA: Freeman, 1979.
[5] E. Horowitz and S. Sahni, "Computing partitions with applications to the knapsack problem," *J. Ass. Comput. Mach.*, Apr. 1974.
[6] D. E. Knuth, *The Art of Computer Programming, Sorting and Searching, Vol. 3.* Reading, MA: Addison-Wesley, 1973.
[7] R. Schroeppel and A. Shamir, "A $TS^2 = O(2^n)$ time/space trade-off for certain NP-complete problems," in *Proc. 20th IEEE Symp. on Foundations of Comput. Sci.*, Oct. 1979.

Ehud D. Karnin (S'80–M'82) was born in Tel-Aviv, Israel, on June 6, 1951. He received the B.S. and M.S. degrees in electrical engineering from the Technion–Israel Institute of Technology, Haifa, Israel, in 1973 and 1976, respectively, and the M.S. degree in statistics and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 1983.

From 1973 to 1979 he was a Research Engineer at Rafael, Israel. From 1980 to 1982 he was a Research Assistant at Stanford University. During 1983 he was a Visiting Scientist at the IBM Research Center, San Jose, CA. He is now a Research Staff Member at the IBM Scientific Center, Haifa, Israel. His research interests are cryptography, information theory, signal processing, and VLSI systems.