A Sparse Knapsack Algo-tech-cuit and its Synthesis*

Rumen Andonov† and Sanjay Rajopadhye‡
IRISA, Campus de Beaulieu
35042 Rennes France

Abstract

We systematically derive an improved algorithm (called the sparse algorithm) for the general knapsack problem which has better average case performance than the standard (dense) dynamic programming algorithm. The derivation is based on transformation of the standard recurrences into a stream functional programs, and cannot be achieved by the usual space-time mapping techniques because the dependencies are statically unpredictable. Furthermore such a sparse algorithm for the general knapsack problem has not been proposed in the literature, to the best of our knowledge. We also implement the sparse algorithm on a linear asynchronous array with constant size memory on each PE (i.e., a wavefront array processor). Using LPGS partitioning, the algorithm can run on an arbitrasy size ring and has optimal time speedup.

1 Introduction

The knapsack problem is a a classic, NP-complete combinatorial optimization problem with many applications [15, 20], and can be formulated as follows: we are given a knapsack of capacity c, into which we may put m types of objects, each object of type i has a *profit*, p_i, and a *weight*, w_i, (w_i, p_i, m and c are all positive integers). Determine z_i, the number of i-th type objects to be chosen so as to maximize the total profit without exceeding the capacity, i.e.,

$$\max \left\{ \sum_{i=1}^{m} p_i z_i \ : \ \sum_{i=1}^{m} w_i z_i \leq c, \ z_i \geq 0 \text{ integer}, \ i = 1, 2, \ldots, m \right\} \tag{1}$$

If z is restricted to be a *binary* vector (i.e., each object can be chosen at most once) we have the well known 0/1 case. The knapsack problem can be solved sequentially in $\Theta(mc)$ time, and this gives it a *pseudo polynomial* complexity [12] since it is polynomial w.r.t. a parameter (which itself, may grow exponentially with input size).

We study dynamic programming [5, 15], one of the standard approaches to solving this problem. The algorithm works in two phases, a forward phase for calculating the *profit* of the optimal solution, and then, a backtracking phase that uses this to construct the solution vector, z. The recurrence equation for the forward phase is given below.

*Supported by the French Coordinated Research Program C^3, and by the Esprit BRA project NANA 2 (No. 6632).
†On leave from Center of Computer Science and Technology, Sofia, Bulgaria
‡Partially supported by NSF Grant No. MIP-910852

$$f_k(j) = \max(f_{k-1}(j), p_k + f_k(j - w_k)) \tag{2}$$

The equation is defined for all $0 < j \leq c, 0 < k \leq m$, and the boundary conditions are, $f_0(j) = f_k(0) = 0$ and $f_k(j) = -\infty$ when $j < 0$; we are interested in finding $f_m(c)$. The recurrence for the 0/1 knapsack problem is only a slight modification (see the subscript of the second argument):

$$f_k(j) = \max(f_{k-1}(j), p_k + f_{\boxed{k-1}}(j - w_k)) \tag{3}$$

As compared to branch and bound (another technique to solve the problem), it is often argued [20] that dynamic programming calculates the function $f_k(j)$ at *all* points and thus *always* pays the worst case price, $\Theta(mc)$, whereas the average case performance of branch and bound algorithms is often much better. However, Horowitz and Sahni [13] have given a dynamic programming algorithm for the 0/1 problem which has much better average case performance. To date, such an improved algorithm has not been developed for the *general* knapsack problem. In this paper, we systematically derive such an algorithm, and then address its implementation on an application specific processor array. Like the Horowitz-Sahni algorithm, our improved algorithm also uses the fact that the cost function is monotonic—hence one needs to calculate it only at certain *critical points*. The contributions of the paper* are summarized as follows:

- An improved algorithm (called the sparse algorithm) for the general knapsack problem which has better average case performance than the standard (dense) dynamic programming algorithm.

- Systematic derivation of the sparse algorithm by transformation of the recurrence of the standard dynamic programming formulation into a stream functional program.

- Modification of the algorithm for bookkeeping information so that the backtracking phase can also be performed sparsely in a very memory efficient manner (because of space limitations, we do not discuss this here. The interested reader is referred to [3], available for anonymous ftp from `ftp.irisa.fr`).

- Implementation of the sparse algorithm on a linear asynchronous array with constant size memory on each PE (i.e., a wavefront array processor [17]) by proving bounds on the size of local buffer memory in each processor.

The remainder of the paper is organized as follows: Sec 2 gives an intuitive explanation of the Horowitz-Sahni 0/1 knapsack algorithm. Then, Sec 3 presents the formal derivation of the sparse algorithm for the general knapsack problem (because of space limitations we only treat the forward phase). Sec 4 shows how the algorithm can be implemented on a wavefront array processor, where each PE has only finite memory. Finally, Sec 5 gives our conclusions and describes open problems. We conclude this introduction with a brief survey.

*algo-tech-cuit = $\boxed{\text{algo}}$-rithm + archi-$\boxed{\text{tech}}$-ture + cir-$\boxed{\text{cuit}}$.

1.1 Related Work

Many authors have investigated parallel (dense) dynamic programming algorithms for the knapsack problem. Most work has concentrated on the 0/1 problem (for example, [10]), and addresses different models of parallel machines—PRAMs, distributed memory machines, linear arrays, meshes and perfect shuffles. There has also been some work on the general knapsack problem, all dealing with the dense dynamic programming algorithm [2, 7, 22]. For this, the best performance is obtained by our systolic algo-tech-cuit [2] (see references therein for a complete discussion).

The basic idea [21] of the sparse algorithm is a quarter century old, but seems to have been used only for the 0/1 problem, to the best of our knowledge. Horowitz and Sahni give sequential algorithms for the 0/1 knapsack and related problems using this idea [14] (Sec 5.5) [13], and Lee et al. implement it on a hypercube using a divide and conquer strategy [18], which takes $O(mc/q + c^2)$ time on q processors, and uses $O(mc)$ storage in the worst case*. Chen et al. present a pipelined linear array which uses $\Theta(mc)$ storage, $\Theta(c)$ on each of m processors [8, 9]. They claim that the backtracking takes $O(m)$ time, but do not give an algorithm to substantiate this claim.

We mention that our model of computation is weaker—an asynchronous ring, where each processor has only a constant memory, and we achieve optimal performance. Unlike much of the theoretical work, we do not attempt to bring the running time down to polynomial at the expense of an exponential number of processors (even with the rapid advances in VLSI technology, we feel that this is too expensive).

2 Intuitive Explanation of the Sparse Algorithm

We now present an intuitive explanation of the Horowitz-Sahni sparse algorithm for the 0/1 knapsack problem. The explanation is substantially the same as the one given in [14] (Sec. 5.5). In the recurrence (3), $f_k(j)$ represents the optimal profit that can be achieved using only the objects $1 \dots k$, and with a capacity of j. If we add more capacity (i.e., as j increases), we can only *improve* the total profit, and we thus have the following observation.

Remark 1 *The function $f_k(j)$ is monotonically increasing in j.*

Now, any monotonic function has a "stepwise" shape (see Fig 1), and hence, for any argument, j, one can determine its value if we know its values at the start of each step. It may therefore be represented completely by a set, S_f of *pairs* (called critical points) $[x, F(x)]$, giving the coordinates where it changes, and the new value.

The sparse algorithm is based on two key observations:

Remark 2 *If $F(x)$ is a monotonic function, then, for positive integers, p and w, the function $p + F(x - w)$ is also monotonic.*

In fact, it is simply the function, $F(x)$ shifted to the right by w and up by p (see Fig 2). As a result, it can also be represented by a set, S'_f of pairs. S'_f is constructed from S_f

*Lin and Storer point out that this could be worse than the sequential algorithm [19]. Note however, that the average behavior is expected to be better because of sparsity.

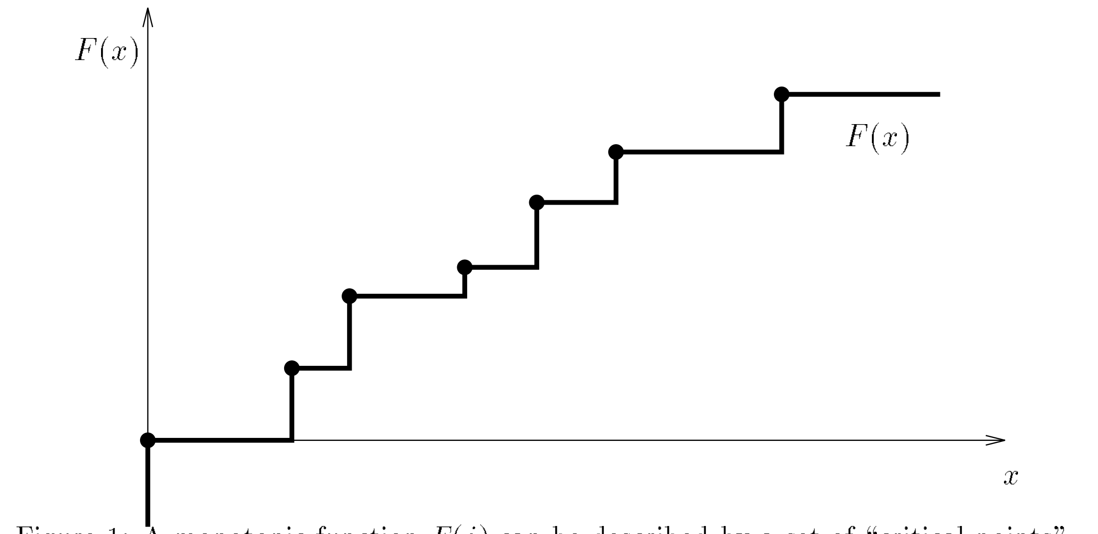

Figure 1: A monotonic function $F(j)$ can be described by a set of "critical points".

by simply adding the pair $[w, p]$ to each element and by including $[w, p]$ as an additional element in the set:

$$S'_f = \{[w, p]\} \cup \{[x + w, F(x) + p] \mid [x, F(x)] \in S_f\}$$

Remark 3 *If $F(x)$ and $G(x)$ are two monotonic functions, then $H(x) = \max(F(x), G(x))$ is also monotonic.*

Fig 3 illustrates this, where $F(x)$ and $G(x)$ are shown in thin lines (their critical points are circles and squares, respectively). The max of the two functions is simply the "outer

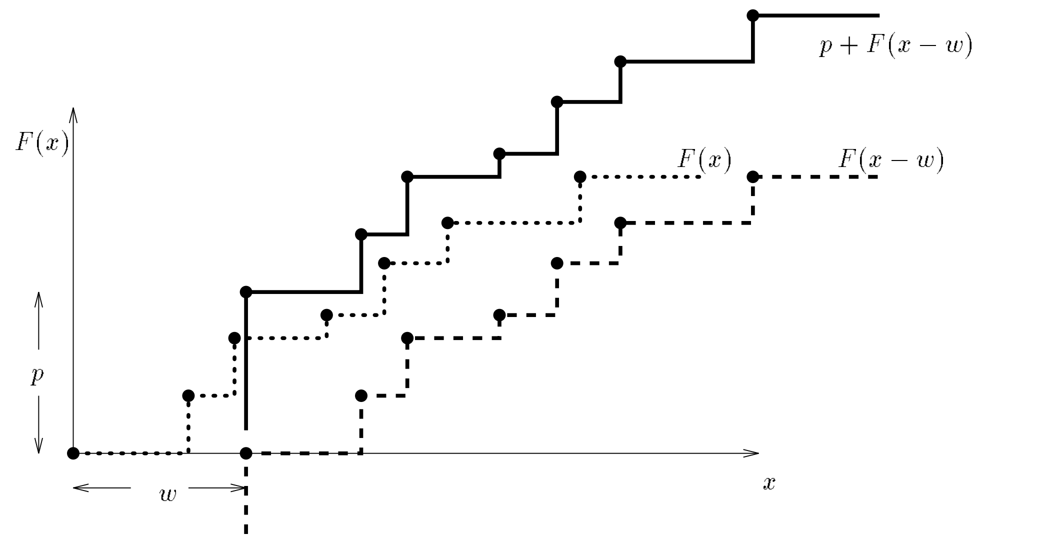

Figure 2: $F(j)$ is monotonic (shown dotted); so are $F(x-w)$—shifted to the right (dashed) as well as $p + F(x - w)$—shifted right *and* up (solid).

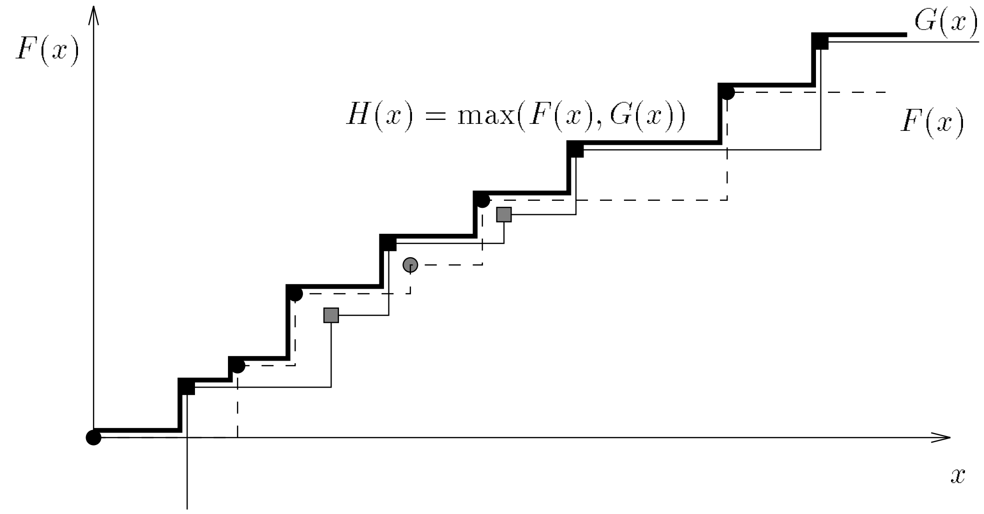

Figure 3: $H(x) = \max(F(x), G(x))$ is also monotonic

envelope" as shown by thick line. $H(x)$ is also monotonic, and its are either the circles or squares (i.e., only critical points of one of the two functions). Note also that some of the critical points (of $F(x)$ as well as $G(x)$) may not appear (shown in gray) in $H(x)$.

Hence we have the following algorithm:

1. $S_1 = \{[w_1, p_1]\}$

2. for $k = 2 \ldots m$, construct S_k as follows:

   (a) Let $S' = S_{k-1} \cup \{[w_k, p_k]\} \cup \{[x + w_k, f + p_k] \mid [x, f] \in S_{k-1} \wedge x + w_k \leq c\}$ (this is a superset of the critical points of S_k).

   (b) Construct S_k by removing all critical points $[x, f] \in S'$ that are *dominated* by another, i.e., if $\exists [x', f'] \in S' \mid x' \leq x$ and $f' \geq f$.

3. Now $f_m(c)$ can be easily obtained from S_m.

3 Formal Derivation of the Sparse Algorithm

Although the basic idea behind the above algorithm is over a quarter century old [21], it has not been applied to the general knapsack problem, to the best of our knowledge. It is clear that Remarks 1–3 are still valid, as illustrated in Tables 1 and 2. However, the presence of f_k on the right hand side of (2) precludes a naive extension, since now, S_k depends (recursively) on itself. Therefore we need to develop an algorithm that constructs it *incrementally* using the partially computed S_k. For this reason, we represent the function as a stream of pairs rather than a set. Furthermore, we maintain two aditional invariants

y k	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	0	0	0	$\boxed{7}$	7	7	7	7	$\boxed{14}$	14	14	14	14
2	0	0	0	$\boxed{8}$	8	8	8	$\boxed{16}$	16	16	16	$\boxed{24}$	24	24
3	0	0	0	$\boxed{8}$	8	$\boxed{9}$	9	$\boxed{16}$	16	$\boxed{17}$	17	$\boxed{24}$	24	$\boxed{25}$
4	0	0	0	$\boxed{8}$	8	$\boxed{9}$	9	$\boxed{16}$	16	$\boxed{24}$	24	24	24	$\boxed{32}$

Table 1: Values of $f_k(j)$ for a simple general knapsack problem: $m = 4$, $c = 14$, $w_i = 5, 4, 6, 10$; $p_i = 7, 8, 9, 24$ showning the monotonicity.

k						
1	[5,7]	[10,14]				
2	[4,8]	[8,16]	[12,24]			
3	[4,8]	[6,9]	[8,16]	[10,17]	[12,24]	[14,25]
4	[4,8]	[6,9]	[8,16]	[10,24]	[14,32]	

Table 2: The pairs used to represent the example of Table 1.

in our representation: the pairs are sorted in increasing order of weights, j, (and hence also the associated, f's); and the j's are bounded by c, the knapsack capacity.

Let, for $k = 1 \ldots m$, `S(k)` be the sequences that represent $f_k(j)$. We now systematically derive a functional program that operates on sequences of pairs and calculates `S(k)` from `S(k-1)`. We use the following notation for writing such programs: sequences are delimited by braces (so, `{}` denotes the empty sequence); `x^S` denotes a sequence whose *head* is the element `x`, and whose *tail* (i.e., the rest of the sequence) is the sequence, `S`; the elements of our sequences in this paper are pairs of integers, `[j,f]` (we use brackets for constructing pairs). Function application uses parentheses (eg, `Merge(A,B)` is the result of applying the function `Merge` to the two arguments `A` and `B`. We define functions equationally, using separate equations for different patterns of the inputs; for example, the following specifies the behavior of the `Merge` function, when either of its inputs is the empty sequence:

```
Merge({}, S) = S
Merge(S, {}) = S
```

Our derivation is based on the following series of lemmata, which follow directly from Remarks 1–3, the discussion in the previous section, and the invariants of our representation.

Lemma 1 *If $f_k(j)$ is represented by the sequence* `S(k)`, *then,* $p_k + f_k(j - w_k)$ *is represented* *by* `AddTest([wk,pk], [0,0]^S(k), c)` *where*

```
AddTest([w,p], {}, c) = {}
AddTest([w,p], [j,f]^S, c) = if j+w > c : {}
                             else       : [j+w, f+p] ^ AddTest([w,p],S,c)
```

Lemma 2 *If $F(x)$ and $G(x)$ are monotonic functions represented by* `S1` *and* `S2`*, respectively, then $H(x) = \max(F(x), G(x))$ is represented by a* **subsequence** *of* `Merge(S1, S2)`*, defined as follows:*

```
Merge({}, S) = S
Merge(S, {}) = S
Merge([j1,f1]^S1, [j2,f2]^S2) =
  if j1 < j2       :  [j1,f1] ^ Merge(S1, [j2,f2]^S2)
  else if j2 < j1  :  [j2,f2] ^ Merge([j1,f1]^S1, S2)
  else if j1 = j2  :  [j1,max(f1,f2)]^Merge(S1, S2)
```

Note that in the last clause above, the head of both sequences is consumed and a single output element is produced. This ensures that the output produced by the `Merge` is *strictly* increasing in its j values, a fact that will be useful later. Thus, the function `Merge` achieves the equivalent of Step 2(a) of the sparse 0/1 algorithm. Now, all that is left is to remove the dominated elements. Recall that a pair $[j', f']$ dominates $[j, f]$ iff $j' \leq j \wedge f' \geq f$, i.e., it has smaller (or equal) weight but larger (or equal) profit. Because the output of `Merge` is in order of increasing weights, the dominator (if any) of an element must be one of its *predecessors in the sequence*. Hence we have the following.

Theorem 1 *The following program computes S_k from S_{k-1}.*

```
S(k) = Filter(Merge(S(k-1), AddTest([wk,pk], [0,0]^S(k), c), 0)
```

where

```
Filter({}, limit) = {}
Filter([j,f]^S, limit) =  if f > limit : [j,f]^Filter(S, f)
                          else         :       Filter(S, limit)
```

Note that `S(k)` occurs on the right hand side, so the program defines the sequence recursively. Furthermore, the structure of the program is a standard one that occurs in lazy functional languages [4, 11], and is often used to manipulate infinite sequences (without having to evaluate them). The final solution of the (forward phase of the) general knapsack problem is obtained from `S(m)`, with the initialization that `S(0)={}`.

4 A Linear Wavefront Array Processor

It is well known [16] that a stream functional program can be implemented as a network of processes, communicating asynchronously through fifo buffers. Such a network can be systematically constructed based only on the *syntactic structure* of the program. For example, the processor that implements our program to compute `S(k)` from `S(k-1)` is

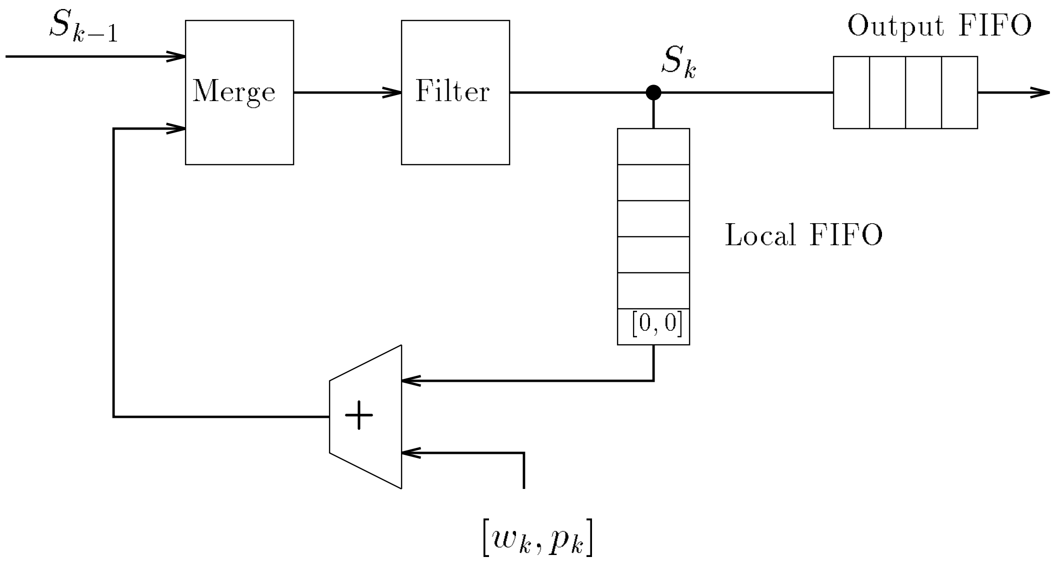

Figure 4: Direct hardware implementation of the program of Theorem 1.

constructed as follows. Our output is the result of applying the function `Filter` (with a scalar parameter, 0) to the output of a `Merge` unit. The latter takes two arguments, one of which is the input sequence, and the other is the output of an `AddTest` unit The input of this is itself the output sequence `S(k)` (with the pair `[0,0]` at its head, giving us a feedback loop in the processor. The feedback loop has a fifo buffer which is initialized with `[0,0]` (see Fig 4).

Looking at the definitions of these three functions, we see that `Filter` can be implemented using a simple "threshold" register which is initialized to 0. If the f value on the input pair is greater than the threshold, it is simply passed through (updating the threshold register to f), otherwise it is simply consumed without producing an output (we say that it is *filtered out*). The `Merge` unit is straightforward, and is implemented with two comparators (the second one is used to compare f values in case the j's are equal). `AddTest` is implemented simply by a pair of adders, and a comparator for generating an "end of sequence" tag.

Now, to implement the recurrence (2), we need m such processors, connected in a linear array (if fewer processors are available, the algorithm can be implemented by configuring them as a ring and using multiple passes). Each processor is connected to the next through an *output buffer*, and has a local feedback buffer described above.

It can be shown that such a network of processors is a *partially correct* implementation of the given program—if the program terminates, then the network of processes also terminates and produces the same values. All our arguments in the previous section dealt with partial correctness of the program, and we need to prove that the program of Theorem 1 does terminate. To show this we need to prove one crucial fact, namely that it is free of deadlock (because of the recursive nature of the program, this is a distinct possibility). We will show this by proving a lower bound on the size of the buffers.

A second important aspect is the memory requirement, i.e., the size of the buffers. For a software implementation on a general purpose machine, this is not a limitation. However, our target is a dedicated VLSI processor, and it becomes crucial to prove that the buffers are bounded from above. The following theorem answers these two problems for the feedback buffer.

Theorem 2 *At any time, the number of elements in the local queue of the k-th processor is bounded from above by w_k, and from below by 1.*

Proof: Let, at any instant, the local buffer contain elements $[j_1, f_1], \ldots [j_l, f_l]$; the buffer length is thus l. Also, let the value in the threshold register of the `Filter` unit be f', and let $[j, f]$ be the value at the head of the input sequence. Now, we have the following four cases.

1. `Merge` selects the local value, $[j_1, f_1]$, which is (a) filtered out, or (b) passed through by `Filter`.
2. `Merge` selects the input value, $[j, f]$, which is (a) filtered out, or (b) passed through by `Filter`.

In case 1.a, $[j_1, f_1]$ is deleted from the front of the buffer, and the size of the local buffer decreases by 1. In case 1.b, $[j_1, f_1]$ is deleted from the front of the buffer, and $[j_1 + w_k, f_1 + p_k]$ is placed at the end, so the buffer size remains l. In case 2.a also, it remains the same (the input value, $[j, f]$ is simply deleted). Finally, in case 2.b it increases by 1 ($[j, f]$ is placed at the end of the buffer, without changing the front). We now consider case 1.a to show a lower bound, and 2.b to show an upper bound.

Now, the fact that the value in the threshold register is f' means that there is a pair $[j', f']$, in the local buffer (which has been passed through some time in the past*). In case 1.a, $f_1 + p_k \leq f'$ (by definition of being filtered out), and thus the pair $[j', f']$ is a *successor* of $[j_1, f_1]$. Hence, whenever the buffer size decreases, there is at least one other value in the local buffer that has not yet been consumed.

Now, for case 2.b, we have the following constraints:

$$\begin{array}{rcl} j_1 + w_k & > & j \\ j & \geq & j_1 + l \end{array}$$

The first constraint is because $[j, f]$ is the winner, while the second is because the output sequence is monotonic and there are $l - 1$ elements between $[j_1, f_1]$ and $[j, f]$. Hence, $l < w_k$. Since the buffer is initialized with the singleton, $[0, 0]$, its length is *always* bounded from above by w_k. ∎

Concerning the size of the output buffer, observe that the processors are connected linearly, only through these buffers. This means that the flow of data is unidirectional. Now, if an output buffer is full, the processor is forced to wait. However, we can show by

*Note that initially, $f' = 0$ and the local buffer is initialized with $[0, 0]$

simple induction that its output buffer will eventually become non-full (assuming that the host is continuously consuming data from the output buffer of the last processor). The processor is thus idle only for a bounded number of steps, and there is no possibility of deadlock. The output buffer size cannot affect the correctness of the implementation, but only the speed (latency). In practice the size can be determined by extensive simulation. Our preliminary results indicate that a very small number (two or three registers) is usually sufficient.

4.1 Implementation with Fixed Size Buffers

From Theorem 2, we now have a linear asynchronous array whose memory is bounded by $w_{\max}$, the largest possible weight, a problem parameter. Because of this, our solution is not completely modular—if $w_{\max}$ increases, it will be necessary to redesign the processor. However, we can use the approach that was previously used for deriving a pure systolic array for the dense knapsack problem [1]. If we have processors with buffer space for only α pairs, we simply configure $\lceil w_i/\alpha \rceil$ physical processors to simulate one processor. Since all the w_i values are known when a problem is loaded, we can also determine (off line) a few bits of configuration information which informs the PE whether it is to participate actively of simply contribute its memory (it can be easily shown that two bits are enough following the lines of [1]). Such an array with q processors, configured into a ring, can solve any problem instance in $\left\lceil \frac{1}{q} \sum_{i=1}^{m} \lceil w_i/\alpha \rceil \right\rceil$ passes.

5 Conclusions

In this paper, we have used some very old ideas to derive a sparse algorithm for the dynamic programming technique for solving the general knapsack problem. The basic idea is the fact that the cost function is monotonic [21], and it suffices if it is calculated only at certain *critical points*, i.e., points where the function changes value. Because of this, the function can be represented as sequences of pairs. The sequences are, on the average, much shorter than the knapsack capacity, and this is the key to the improvement of the average running time. Horrowithz and Sahni have presented extensive empirical data to support this fact for the sparse 0/1 algorithm, and our preliminary simulations confirm that this is also true for the general problem. An analytic study of the average case complexity is the subject of our ongoing work.

Another contribution of the paper is the systematic derivation of the algorithms by transforming the original recurrence equation into stream functional programs. Our derivation is almost mechanical, and provides a proof of correctness of the derived programs. Indeed, we feel that it is our systematic derivation that clarifies the behavior of the algorithms.

Finally, we have shown how the stream functional programs that we derived could be implemented on a wavefront array processor, an asynchronous processor array, where each PE has fixed memory, independent of problem parameters. In such an array, problems with larger parameters sizes are simply solved by adding more processors, or reusing the same array in multiple passes. In order to do this we prove that the lengths of the buffers in architecture is always bounded. This raises an interesting open question. It is well known that lazy stream functional programs [11], correspond almost mechanically to networks of processes communicating through fifo channels [16], these are not realistic because the problem of proving bounds on buffer lengths is undecidable [23]. We were successful in proving the bounds for our example by hand, and it would be interesting to see if the ideas used in the proof could be generalized to certain cases of asynchronous processor arrays.

Finally, it is necessary to analytically determine a good model for the average case behavior of the sparse algorithms and hence the processor arrays. If a closed form for this can be determined, one could try to optimize the expected running time by appropriately choosing the buffer size in each PE. This approach has been successfully used for the dense algo-tech-cuit [2], and it would be interesting to extend it to the sparse case.

Acknowledgements The authors would like to thank Doran Wilde for many helpful discussions, and for much help in simulating the algorithms. Sincere thanks to Paul Caspi for pointing out deadlock problems with a preliminary version of the algorithm, and also to the anonymous referees for suggestions that substantially improved (we hope) the presentation.

References

[1] R. Andonov, P. Quinton, S. Rajopadhye, and D. Wilde. *A Shift-Register based Systolic Array for the General Knapsack Problem.* Technical Report PI-813, IRISA, Rennes, France, March 1994. (submitted to Parallel Processing Letters; a preliminary version to appear in PARCELLA 94.

[2] R. Andonov and S. Rajopadhye. *An Optimal Algo-tech-cuit for the Knapsack Problem.* Technical Report PI-791, IRISA, Campus de Beaulieu, Rennes, France, January 1994. (submitted to IEEE Transactions on Parallel and Distributed Systems).

[3] R. Andonov and S. Rajopadhye. *A Sparse Knapsack Algo-tech-cuit and its Synthesis.* Technical Report PI-801, IRISA, Rennes, France, February 1994.

[4] E. Ashcroft and W. Wadge. LUCID, A formal system for writing and proving programs. *SIAM Journal on Computing*, 3:336–354, 1976.

[5] R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[6] P. Caspi. Lucid synchrone. In *Proc. OPOPAC*, pages 79–93, Hermes, Paris, 1993.

[7] G.H. Chen, M.S. Chern, and J.H. Jang. Pipeline architectures for dynamic programming algorithms. *Parallel Computing*, 13:111–117, 1990.

[8] G.H. Chen and J.H. Jang. An improved parallel algorithm for 0/1 knapsack problem. *Parallel Computing*, 18:811–821, 1992.

[9] G.H. Chen and M.S.Chern. A pipelined algorithm for multiple-choice 0/1 knapsack problem. *International Journal of High Speed Computing*, 4(1):43–48, 1992.

[10] A. G. Ferreira. The knapsack problem on parallel architectures. In M. Cosnard, Y. Robert, P. Quinton, and M. Raynal, editors, *International Workshop on Parallel and Distributed Algorithms*, pages 145–152, North Holland, Bonas, France, October 1988.

[11] D. Friedman and D. Wise. CONS should not evaluate its arguments. In *ICALP'76*, pages 257–284, 1976.

[12] M. Garey and D. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness.* Freeman, San Francisco, 1979.

[13] E. Horowitz and S. Sahni. Computing partitions with aplications to the knapsack problem. *Journal of the ACM*, 21(2):277–292, April 1974.

[14] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, Maryland, USA, 1978.

[15] T. C. Hu. *Integer Programming and Network Flows.* Addison-Wesley, 1969.

[16] G. Kahn. The semantics of a simple language for parallel processing. In *Proceedings of IFIP*, pages 471–475, IFIP, August 1974.

[17] S. Y. Kung, K. S. Arun, R. J. Gal-Ezer, and D. V. B. Rao. Wavefront array processor: language, architecture and applications. *IEEE Transactions on Computers*, C-31:1054–1066, 1982.

[18] J. Lee, E. Shragowitz, and S. Sahni. A hypercube algorithm for the 0/1 knapsack problems. *J. of Parallel and Distributed Computing*, 5:438–456, 1988.

[19] J. Lin and J. A. Storer. Processor-efficient hypercube algorithm for the knapsack problem. *Journal of Parallel and Distributed Computing*, 13:332–337, 1991.

[20] S. Martello and P Toth. *Knapsack Problems: Algorithms and Computer Implementation.* John Wiley and Sons, 1990.

[21] G. Nemhauser and J. Ullman. Discrete dynamic programming and capital allocation. *Management Science*, 15(9):494–505, 1969.

[22] S. Teng. Adaptive parallel algorithm for integral knapsack problems. *Journal of Parallel and Distributed Computing*, 8:400–406, 1990.

[23] P. Wadler. Listlessness is better than lazyness: lazy evaluation and garbage collection at compile time. In *Proceedings: Symposium on LISP and Functional Programming*, pages 45–52, ACM, 1984.