

An Alternative Dynamic Programming Solution for the 0/1 Knapsack

Timothy J. Rolfe

Department of Computer Science
Eastern Washington University
319F Computing & Engineering Bldg.
Cheney, Washington 99004-2493 USA
Timothy.Rolfe@mail.ewu.edu

Abstract: The 0/1 knapsack (or knapsack without repetition) has a dynamic programming solution driven by a table in which each item is consecutively considered. The problem can also be approached by generating a table in which the optimal knapsack for each knapsack capacity is generated, modeled on the solution to the integer knapsack (knapsack with repetition) found in Sedgewick [1] and the solution to change-making found in Ciubatii [2].

Categories and Subject Descriptors: F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems---Computations on discrete structures; G.2.3 [Discrete Mathematics]: Applications

General Terms: Algorithms

Keywords: Knapsack, 0/1 Knapsack, Knapsack without Repetition, Dynamic Programming, Algorithm Analysis

1. LITERATURE SOLUTION

While Corman and others [3] note that the 0/1 knapsack can be solved through dynamic programming, they relegate that solution to an exercise. Brassard and Bratley [4] and Levitin [5] present a bottom-up table-driven dynamic programming solution to the problem: build a table with $(nItems+1)$ rows and $(capacity+1)$ columns. From item #1 (the zeroeth row is left as a sentinel row) on down, obtain the values of knapsacks of the weights from 1 up to the capacity using only items from item #1 down to the current row. Once the table is completed, the solution can be obtained by examining the table, starting from the $[nItems][capacity]$ cell. Levitin [5, p. 303] shows the following table as Figure 8.13.

		capacity j						
		j	0	1	2	3	4	5
$w_1 = 2, v_1 = 12$	0	0	0	0	0	0	0	0
$w_2 = 1, v_2 = 10$	1	0	0	12	12	12	12	12
$w_3 = 3, v_3 = 20$	2	0	10	12	22	22	22	22
$w_4 = 2, v_4 = 15$	3	0	10	12	22	30	32	32
	4	0	10	15	25	30	37	37

See the referenced texts for the rules for filling the table, and then for traversing the table to obtain the contents of the optimal knapsack.

2. MODELS FOR AN ALTERNATIVE SOLUTION

Sedgewick [1] presents a recursive dynamic programming solution to the integer knapsack problem (or knapsack with repetition) using memoization or a "memory function". The global vector `maxKnown[]` contains the value for the knapsack with the weight represented by its subscript. Generation of the knapsack is driven by the global vector `itemKnown[]`. The recursive solution is started by invoking the function for the capacity of the knapsack. In the recursion, in the standard form for a memory function solution, if the current cell of the global vector shows it as not yet computed, it is filled through recursive calls for lower subscripts. In this case, searching for the best item to optimize the value for this knapsack capacity and then using it. [There is a small bug, in that the function as presented fails to find a solution if the available items cannot completely fill the knapsack. Christopher Coleman, an Eastern Washington University undergraduate student, suggests supplementing the items with one of weight one and value zero to insure a solution.]

This recursive memory function implementation could also be the basis of an iterative bottom-up implementation simply by filling the vectors beginning at weight one and working up to the knapsack capacity. This populates all cells, while the memory function implementation only populates cells required for the final solution.

Such a bottom-up implementation is suggested by the solution to the change problem presented by the TopCoder columnist Dumitru Ciubatii for the change-making

problem [2]. He presents the following pseudocode solution.

```

Set Min[i] equal to Infinity for all of i
Min[0]=0

For i = 1 to S
  For j = 0 to N - 1
    If (Vj≤i AND Min[i-Vj]+1<Min[i])
      Then Min[i]=Min[i-Vj]+1

Output Min[S]

```

The merge of Sedgewick's algorithm and Ciubatii's algorithm generates a solution for the integer knapsack whose Java encoding follows (compressed for space):

```

static int knap(int c, int[] solution)
{ int i, space, m, t,
  maxKnown[], itemKnown[];

  itemKnown = new int [c+1];
  maxKnown = new int [c+1];

  // If Java didn't fill with zeroes we'd need
  // java.util.Arrays.fill(maxKnown,0);

  for ( m = 1; m <= c; m++ )
    for ( i = 1; i <= n; i++ )
      if ( (space = m-weight[i]) >= 0 )
        { t = maxKnown[space] + profit[i];
          // Dummy: wt 1, value 0
          if ( t >= maxKnown[m] )
            { maxKnown[m] = t;
              itemKnown[m] = i;
            }
        }

  // State of solution[] is unknown, so

  java.util.Arrays.fill (solution, 0);
  for (m=c; m>0 ; m-=weight[itemKnown[m]] )
    solution[itemKnown[m]]++;
  return maxKnown[c];
}

```

Unlike the bottom-up table-driven solution that corresponds with the 0/1 Knapsack, this solution uses space proportional just to the capacity, rather than the capacity times the number of items.

3. APPLICATION TO THE 0/1 KNAPSACK

The solution for the 0/1 knapsack is represented not as a vector containing the number of instances used for each available item but as a boolean vector indicating whether or not the available item was used. In trying to apply the above approach to the 0/1 knapsack, the `itemKnown[]` vector changes into a matrix of boolean solution vectors, one for each knapsack capacity. The bottom row of the resulting matrix is the solution vector for the problem.

To minimize the number of vector copies, however, one can search through the items to find the optimal item that maximizes the value. For each item k , first check it against the row index: would using this item exceed the capacity of the knapsack? If that is not the case, then check to see whether the lighter knapsack to which we are potentially adding this item is already using it. If not, we can get the value of the combined knapsack and check it against the best found so far.

```

if (bestVal[wt] < value[k]
    + bestVal[wt - weight[k]])
{ bestK = k;
  bestVal[wt] = value[k]
  + bestVal[wt - weight[k]];
}

```

A significant issue, however, is the initial value for each `bestVal[wt]`. As noted above, for a knapsack weight that cannot be achieved with the items available, the optimal knapsack will be the one whose weight is one less. Then that solution vector would also be copied as the current weight's solution vector. Andrew Cobb, an Eastern Washington University graduate student, suggests that one can also use `bestVal[wt-1]` as the initial `bestVal[wt]`. It is possible that the knapsack achieved by forcing a given `wt` might have less value than the `[wt-1]` knapsack. If, then, the search for `bestK` fails (i.e., it is still 0), then the `[wt-1]` solution vector can be copied as the `[wt]` solution. Otherwise the `[wt-weight[bestK]]` vector provides the `[wt]` vector provided one sets to boolean true the `[bestK]` bit. The following Java code segment gives this logic, where `trial` is the matrix of trial solution vectors.

```

for (wt = 1; wt <= maxWeight; wt++)
{ int bestK = 0, testWt;

  // Initial guess: the knapsack for wt-1.
  bestVal[wt] = bestVal[wt-1];
  for (k = 1; k <= n; k++)
    { testWt = wt - weight[k];
      if (testWt >= 0 && ! trial[testWt][k])
        if ( bestVal[wt] <
            value[k]+bestVal[testWt] )
          { bestK = k;
            bestVal[wt] = value[k]
              + bestVal[testWt];
          }
    }
  if (bestK > 0)
    { testWt = wt - weight[bestK];
      System.arraycopy(trial[testWt], 0,
        trial[wt], 0, n+1);
      trial[wt][bestK] = true;
    }
  else // Finish using the wt-1 solution
    System.arraycopy(trial[wt-1], 0,
      trial[wt], 0, n+1);
}

```

Running this algorithm on the data represented by Levitin's table above gives the following results.

Item list	wt	Trial Solutions					bestVal
		Item k					
		0	1	2	3	4	
$w_1 = 2, v_1 = 12$	0	0	0	0	0	0	0
$w_2 = 1, v_2 = 10$	1	0	0	1	0	0	10
$w_3 = 3, v_3 = 20$	2	0	0	0	0	1	15
$w_4 = 2, v_4 = 15$	3	0	0	1	0	1	25
	4	0	0	1	1	0	30
	5	0	1	1	0	1	37

While one could turn this into a memory-function recursive implementation, the check for each w_t-1 within the w_t solution would force the complete filling of the

solution matrix anyway. In such a case, iteration is preferable to recursion. Another advantage of the iterative solution is that it directly generates the solution vector, so that one does not need to traverse the grid to build the solution vector.

4. WEB RESOURCE

<http://penguin.ewu.edu/~trolfe/Knapsack01/index.html>.

This page provides access to this paper, and also to the Java and C implementations of this algorithm, embedded in test programs, along with several specimen data files.

5. ACKNOWLEDGEMENTS

I wish to acknowledge and thank Andrew Cobb and Christopher Coleman, the Eastern Washington University students whose suggestions have been incorporated into this paper.

REFERENCES

- [1] Robert Sedgewick, *Algorithms in C* (3rd edition; Addison-Wesley, 1998), p. 215 as Program 5.13. Also in the same author's *Algorithms in Java* (3rd edition; Addison-Wesley, 2003), p. 225 as Program 5.13.
- [2] Dumitru Ciubatii, "Dynamic Programming: From novice to advanced", URL as of 2007 February 22: <http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=dynProg>.
- [3] Thomas H. Corman, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms* (2nd edition; MIT Press, 2002), pp 382, 384ex.
- [4] Giles Brassard and Paul Bratley, *Fundamentals of Algorithmics* (Prentice-Hall Inc., 1996), pp. 266-68.
- [5] Anany Levitin, *Introduction to the Design & Analysis of Algorithms* (2nd edition; Pearson Education Inc., 2007), pp. 299-303. In the 1st edition (2003), pp. 295-299.

Check out the

AIS

Association for Information Systems

<<http://plone.aisnet.org/>>