EE677 Foundations of VLSI CAD Course Project

Tejas Nikumbh [10d070019]

Project Topic : Dynamic Programming Algorithm for Knapsack problem(0/1).

Problem Introduction:

Problem Genre : Combinatorial Problem

Brief Summary of the problem : Given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

Applications of the problem : The problem often arises in resource allocation where there are financial constraints and is studied in fields such as combinatorics, computer science, complexity theory, cryptography and applied mathematics.This problem was one of the top 5 most needed algorithmic problems according to a survey conducted by Stony Brook University(1998).

Problem Definition:

We will try to solve the 0-1 knapsack problem here, or the binary knapsack problem.

Mathematically the 0-1-knapsack problem can be formulated as:

Let there be $n$items, $x_1$to $x_n$where $x_i$has a value $v_i$and weight w_i. The maximum weight that we can carry in the bag is *W*. It is common to assume that all values and weights are nonnegative. We also assume that the weights are in sorted order(We perform the sorting in the scilab implementation)

- Maximize $\sum_{i=1}^{n} v_i x_i$ subject to $\sum_{i=1}^{n} w_i x_i \leqslant W, \qquad x_i \in \{0,1\}$

We need to maximize the sum of the values subject to the constraint that the weight sum is less than the specified limit.

Proposed Dynamic Programming Algorithm :

A dynamic algorithm for the binary knapsack problem runs in pseudo polynomial time as opposed to the exponential brute force method. So , a dynamic approach to this problem is definitely beneficial. The algorithm can be summarized as follows. Assume

$w_1,\ w_2,\ \ldots,\ w_n,\ W$ are strictly positive integers. Define $m[i,\ w]$ to be the maximum value that can be attained with weight less than or equal to w using items up to i.

We can define $m[i,\ w]$ recursively as follows:

- $m[i,\ w] = m[i-1,\ w]$ if $w_i > w$ (the new item is more than the current weight limit)
- $m[i,\ w] = \max(m[i-1,\ w],\ m[i-1, w-w_i] + v_i)$ if $w_i \leqslant w$.

The solution can then be found by calculating $m[n,\ W]$. To do this efficiently we can use a table to store previous computations.

The recursive relation is logical obvious. If a new weight being introduced is larger than the weight limit, it obviously cannot be included and hence the total weight and hence the value in our knapsack remains the same. If , however, the weight of the new object that can be added to our knapsack, we check two cases, once when it is added and maximum value is calculated, and other when it is not added and maximum value is calculated, and take the maximum out of the two, which corresponds to the max case above.

Pseudo Code (Includes Input and Output Specification):

```
// Note that we assume here that the array is sorted according to weights.
// Input:
// Values (stored in array v)
// Weights (stored in array w)
// Number of distinct items (n)
// Knapsack capacity (W)
// Output:
// m[n,W] - The maximum value that can be accumulated

for w from 0 to W do
  m[0, w] := 0
end for
for i from 1 to n do
  for j from 0 to W do
    if j >= w[i] then
      m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])
    else
      m[i, j] := m[i-1, j]
    end if
  end for
end for
```

Time Complexity and space complexity:

This solution will therefore run in O(n*W)time and O(n*W)space. Additionally, if we use only a 1-dimensional array m[w] to store the current optimal values and pass over this array (i+1) times, rewriting from m[w]to m[1]every time, we get the same result for only O(W)space.

Worked Out Example:

Consider the case where W = 15kg, and in sorted order, the xi's have weights and values{ 1,1} ,{ 1,2}, {2,2},{4,10},{12,4}. First one is weight and second one is value.

Applying the above algorithm we get the following pseudo code.

```
//w is array of weights where,
 w[1]=1,w[2]=1,w[3]=2,w[4]=4,w[5]=12
//v is array of respective values
 v[1]=1,v[2]=2,v[3]=2,v[4]=10,v[5]=4
//Assigning m[0,w] for all w from 0 to 15 to be 0
for w from 0 to 15 do
  m[0, w] := 0
end for
for i from 1 to 5 do
  for j from 0 to 15 do
    if j >= w[i] then
      m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])
    else
      m[i, j] := m[i-1, j]
    end if
  end for
end for
Resulting table of m's is as follows.
The rows are the (i's).
The columns are the (j's).
On the intersection you find the values of corresponding m[i,j]'s.
```

i/j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3
3	0	2	3	4	5	5	5	5	5	5	5	5	5	5	5	5
4	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

The final answer according to the above table is **$15**. That is , you can fill your sack up with items whose maximum value is $15 according to the above table, if the maximum capacity of your rucksack is 15kg.

Also, it is clear that the **time complexity because of the two for loops is n*W which is O(nW)**. The **space complexity is clearly O(n*W)**, as **the table drawn above** to store the values indicates. If we need a **space complexity of O(W**), that is also possible if **we re-write the m array each time we loop** over the j's or the weights, storing only the recent most ith row.

References : Wikipedia