Department of
Computer Science and Engineering

Title: Implement 0-1 Knapsack Problem Using Dynamic Programming (DP)

Algorithms Lab
CSE 206

Green University of Bangladesh

1 Objective(s)

- Understand the basic of dynamic programming
- Apply dynamic programming to solve real-life optimal decision making

2 Problem Analysis

Suppose a thief is going to steal a store. He has a knapsack to carry goods and maximal weight of W is possible to carry. There are n items available in the store and weight of i-th item is w_i and its profit is p_i. What items should the thief take? Problem is he have to take the item entirely or left it behind which is denoted by $x_i = 0, 1$. Therefore, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit. Hence, the objective of the thief is to maximize the profit –

$$\max \sum_{i=1}^{N} x_i p_i \tag{1}$$

In addition, the constraint is-

$$\sum_{i=1}^{N} x_i w_i \leq W \tag{2}$$

2.1 Solution Steps

- Take input of list of items, and weights using array
- Construct a DP table $P[n][W]$, where $P[i][w]$ indicates the maximum profit that can be obtained from items 1 to i, if the knapsack has size w
- Case 1: taking the item i, in that case-
  $P[i][w] = v_i + P[i-1][w-w_i]$
- Case 2: not taking the item i, in that case-
  $P[i][w] = P[i-1][w]$
- The final recurrence relation is-
  $P[i][w] = max\{v_i + P[i-1][w-w_i], P[i-1][w]\}$

We can understand the problem more clearly by the following example

Item	Weight	Value
1	2	12
2	1	10
3	3	20
4	2	15

Figure 1: Weight and profit of each items

i \ w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	12	12	12	12	12
2	0					
3	0					
4	0					

P[1][1] = P[0][1] = 0
P[1][2] = max{ 12+0, 0} = 12
P[1][3] = max{ 12+0, 0} = 12
P[1][4] = max{ 12+0, 0} = 12
P[1][5] = max{ 12+0, 0} = 12

Figure 2: Iteration 1

i \ w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	12	12	12	12	12
2	0	10	12	22	22	22
3	0					
4	0					

P[2][1] = max{ 10+0, 0} = 10
P[2][2] = max{ 10+0, 12} = 12
P[2][3] = max{ 10+12, 12} = 22
P[2][4] = max{ 10+12, 12} = 22
P[2][5] = max{ 10+12, 12} = 22

Figure 3: Iteration 2

Calculate the Iteration 3 by yourself

i \ w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	12	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

P[4][1] = P[3][1] = 10
P[4][2] = max{ 15+0, 12} = 15
P[4][3] = max{ 15+10, 22} = 25
P[4][4] = max{ 15+12, 30} = 30
P[4][5] = max{ 15+22, 32} = 37

Figure 4: Iteration 4

2.2 Time Complexity

Time complexity of $0-1 Knapsack$ problem is $O(nW)$ where, n is the number of items and W is the capacity of knapsack.

3 Algorithm

Algorithm 1: Dynamic 0-1 Knapsack

Input: Weights, Values
Output: $P[n, W]$

for $w = 0$ *to* W **do**
  $P[0, w] = 0$
end
for $i = 1$ *to* n **do**
  $P[i, 0] = 0$
  for $w = 1$ *to* W **do**
    if $w_i \leq w$ **then**
      $P[i, w] = max\{v_i + P[i-1, w-w_i], P[i-1, w]\}$
    end
    else
      $P[i, w] = P[i-1, w]$
    end
  end
end

4 Implementation in Java

```java
import java.util.Scanner;
// A Dynamic Programming based solution
// for 0-1 Knapsack problem
public class Knapsack {
  // A utility function that returns
  // maximum of two integers
  static int max(int a, int b)
  {
    return (a > b) ? a : b;
  }
  // Returns the maximum value that can
  // be put in a knapsack of capacity W
  static int knapSack(int W, int wt[],
            int val[], int n)
  {
    int i, w;
    int P[][] = new int[n + 1][W + 1];

    // Build table K[][] in bottom up manner
    for (i = 0; i <= n; i++)
    {
      for (w = 0; w <= W; w++)
      {
        if (i == 0 || w == 0)
          P[i][w] = 0;
        else if (wt[i - 1] <= w)
          P[i][w]
            = max(val[i - 1]
            + P[i - 1][w - wt[i - 1]],
            P[i - 1][w]);
        else
          P[i][w] = P[i - 1][w];
      }
    }

```

```
        return P[n][W];
    }
    // main method
    public static void main(String args[]) {
          Scanner sc = new Scanner(System.in);
          System.out.println("Enter No. of Items");
          int n = sc.nextInt();
          System.out.println("Enter size of Knapsack");
          int W = sc.nextInt();
          int val[] = new int[n];
          int wt[] = new int[n];
          System.out.println("Enter the values of items");
          for (int i = 0; i < n; i++) {
              val[i] = sc.nextInt();
          }
          System.out.println("Enter the weights of items");
          for (int i = 0; i < n; i++) {
              wt[i] = sc.nextInt();
          }
          System.out.println("Maximum total profit = " + knapSack(W, wt, val, n));
      }
}
```

4.1 Sample Input/Output (Compilation, Debugging & Testing)

Output:
Enter No. of Items
4
Enter size of Knapsack
5
Enter the values of items
12 10 20 15
Enter the weights of items
2 1 3 2
Maximum total profit = 37

5 Discussion & Conclusion

Based on the focused objective(s) to understand the dynamic programming solution of rock climbing, the additional lab exercise will increase confidence towards the fulfilment of the objectives(s).

6 Lab Task (Please implement yourself and show the output to the instructor)

1. Implement Longest increasing sub-sequence problem using DP technique.
    - Hint: {9, 2, 5, 3, 7, 11, 8, 10, 13, 6} is a sequence. A possible longest sub-sequence in increasing order can be {2, 5, 7, 8, 10, 13}

7 Lab Exercise (Submit as a report)

- Given a list of coins i.e 1 taka, 5 taka and 10 taka, can you determine the total number of combinations of the coins in the given list to make up the number N taka?

8 Policy

Copying from internet, classmate, seniors, or from any other source is strongly prohibited. 100% marks will be *deducted* if any such copying is detected.