Yogendra Pokhrel

617585

Q1. Consider the Knapsack problem algorithm we covered in class i.e.

max {V[i-1,j], vi + V[i-1,j- wi]} if j- wi 0

V[i,j] =

V[i-1,j] if j- wi < 0

Initial conditions: V[0,j] = 0 and V[i,0] = 0.

This does not list the items selected when a final optimized solution is obtained.

Use this algorithm and modify it by adding a new array Keep[i,w] which will keep

track of the items selected. Write a pseudo code for this updated algorithm. Also,

make sure you print the results showing items selected [Hint: use similar approach

as in Print-Cut_Rod_Soln].

The given algorithm is for solving the 0/1 Knapsack Problem using dynamic programming. While the base algorithm computes the maximum value achievable within a given weight limit, it does not track which items were selected. To address this, I have modified the algorithm by introducing a 2D array Keep[i][j] that records whether an item is included in the optimal solution. Below is the updated pseudocode and the logic to reconstruct the selected items:

Modified Knapsack Algorithm with Keep Array

Input:

n ← number of items

W ← maximum weight capacity of the knapsack

v[1..n] ← array of item values

w[1..n] ← array of item weights

Output:

V[n][W] ← maximum value

List of selected items

Algorithm:

1. Initialize DP table V[0..n][0..W] to 0

2. Initialize Keep[0..n][0..W] to 0

3. for i from 1 to n:

4. for j from 0 to W:

5. if w[i] ≤ j:

6. if v[i] + V[i-1][j - w[i]] > V[i-1][j]:

7. V[i][j] ← v[i] + V[i-1][j - w[i]]

8. Keep[i][j] ← 1

9. else:

10. V[i][j] ← V[i-1][j]

11. Keep[i][j] ← 0

12. else:

13. V[i][j] ← V[i-1][j]

14. Keep[i][j] ← 0

// Reconstruct selected items

15. j ← W

16. for i from n down to 1:

17. if Keep[i][j] == 1:

18. print("Item", i, "selected (value:", v[i], "weight:", w[i], ")")

19. j ← j - w[i]

Here:

- V[i][j] stores the maximum value for the first i items and capacity j.
- Keep[i][j] = 1 means item i was included in the optimal solution for capacity j.
- At the end, we trace back from V[n][W] using the Keep array to identify the items that were selected.

This approach is inspired by similar logic used in Print-Cut_Rod_Soln where reconstruction of the solution path is required after computing optimal values.

Q2. Determine the running time complexity of your pseudo code. Has Dynamic Programming approach of solving Knapsack problem changed the exponential time complexity of the original brute force solution?

The running time complexity of the modified dynamic programming (DP) algorithm for the 0/1 Knapsack Problem is:

Time Complexity: O(n × W)

Where:

- n is the number of items
- W is the maximum capacity of the knapsack

This is because the algorithm fills a 2D table V[i][j] of size (n+1) × (W+1) and each cell is computed in constant time, using previously computed values.

Comparison to Brute Force Approach:

The brute-force solution tries all possible subsets of items to find the optimal one. Since each item has two possibilities (included or not included), the total number of combinations is:

Exponential Time: $O(2^n)$

Impact of Dynamic Programming:

Yes, the dynamic programming approach dramatically improves the time complexity from exponential $O(2^n)$ in the brute-force method to polynomial O(n × W). This makes it feasible to solve reasonably large instances of the knapsack problem efficiently.

Thus, dynamic programming transforms an otherwise computationally infeasible problem into a solvable one for practical sizes of n and W.

Q3. Use the answer of Q2 to explain how Knapsack problem is exponential (Hint - consider both inputs and see how they [or at least one] can cause Knapsack problem to be exponential)

The Knapsack problem becomes exponential in complexity due to the nature of its input size and how it influences the total number of possible combinations of items.

Let's consider the two inputs to the Knapsack problem:

- n: the number of items
- W: the maximum capacity of the knapsack

In the brute-force approach, for each item, we have two choices: either include it or exclude it. This leads to 2^n possible subsets of items. Therefore, the time complexity is:

$O(2^n)$

– Exponential in the number of items

This is what makes the original Knapsack problem computationally hard for large values of n.

Role of W (Weight Capacity):

Although the weight capacity W is just a numeric input, it becomes part of the state space in the dynamic programming solution. This means the DP algorithm has to compute results for all combinations of n items and W capacities, giving a time complexity of:

$O(n \times W)$

While this is polynomial in the numeric value of W, it is not polynomial in the number of bits required to represent W, which is only $\log_2(W)$ bits. Hence, the DP solution is pseudo-polynomial.

Hence:

The Knapsack problem is considered exponential in the worst case because:

- The brute-force solution examines all 2^n combinations.
- Even the DP solution, though faster, is not fully polynomial in input size (since W is a number, not its bit-length).

Thus, the problem remains NP-complete.