Knapsack Problem (Dynamic Programming)

Topics: #dynamic-programmming
LinkedIn: Md. Ziaul Karim
Find me here:

Contents

Problem Statement

Note: This is an exercise from *Chapter 9: Dynamic Programming* of a book named:
Grokking Algorithms: An Illustrated Guide for Programmers and Other Curious People by Aditya Y. Bhargava.

Even though the book doesn't provide coding solution for this particular problem, here's my attempt to solve this problem.

9.2 Suppose you're going camping. You have a knapsack that will hold 6 lb, and you can take the following items. Each has a value, and the higher the value, the more important the item is:

• Water, 3 lb, 10
• Book, 1 lb, 3
• Food, 2 lb, 9
• Jacket, 2 lb, 5
• Camera, 1 lb, 6
What's the optimal set of items to take on your camping trip?

Next Page

Solution 1: Two Part Solution(dynamic programming w/ backtracking)

The code initializes a 2D array `dp` to store intermediate results. It iterates through each item and weight capacity combination, considering whether it is better to include the current item or not. The result is stored in the `dp` array.

After computing the dynamic programming table, the code backtracks to find the selected items that contribute to the maximum total value, considering the weight constraint.

Code

```
class DynamicProgramming:
    def camping_with_knapsack(self, items,total_capacity, number_of_items):
        dp = [[0]*(total_capacity+1) for _ in range(number_of_items+1)]

                # going through each cell of the table
        for row in range(1, number_of_items+1):
            for col in range(1, total_capacity+1):
                if items[row-1]['weight']<=col:
                    dp[row][col]=max(dp[row-1][col], dp[row-1][col-items[row-1]['weight']]+items[row-1]
['value'])
                else:
                    dp[row][col] = dp[row-1][col]

                #bottom right corner of the table represents total_value
        total_value = dp[row][col]

        #backtracking to see which items were chosen
        row, col = number_of_items, total_capacity
        selected_items = []

        while 0<row and 0<col:
            if dp[row][col] != dp[row-1][col]:
                selected_items.append(items[row-1]) #item selected
                col-=items[row-1]['weight'] # represents remaining capacity
            row-=1 # once an item has been selected, we go up a row
        return total_value, selected_items

if __name__ == '__main__':
    items = [
    { 'name': 'Water', 'weight': 3, 'value': 10 },
    { 'name': 'Book', 'weight': 1, 'value': 3 },
    { 'name': 'Food', 'weight': 2, 'value': 9},
    { 'name': 'Jacket', 'weight': 2, 'value': 5},
    { 'name': 'Camera', 'weight': 1, 'value': 6}
    ]
    total_capacity = 6
    number_of_items = len(items)
    dp = DynamicProgramming()
    total_value, result = dp.camping_with_knapsack(items, total_capacity, number_of_items)

    print("Optimal Set of items to take:\n")
    for item in result:
        print(f"{item['name'].title()} - Weighs: {item['weight']} lbs, Value: ${item['value']}")

    print(f"Total Value: ${total_value}")
```

Next Page

Output

```
# creates
dp = [
 [0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0]
 ]

# after being populated
dp=[
 [0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 10, 10, 10, 10],
 [0, 3, 3, 10, 13, 13, 13],
 [0, 3, 9, 12, 13, 19, 22],
 [0, 3, 9, 12, 14, 19, 22],
 [0, 6, 9, 15, 18, 20, 25]
 ]

#returns
25, [
 {'name': 'camera', 'weight': 1, 'value': 6},
 {'name': 'food', 'weight': 2, 'value': 9},
 {'name': 'water', 'weight': 3, 'value': 10}]

# prints out
Optimal Set of items to take:

Camera - Weighs: 1 lbs, Value: $6
Food - Weighs: 2 lbs, Value: $9
Water - Weighs: 3 lbs, Value: $10
Total Value: $25
```

Next Page

Solution 2: The lazy Solution

In this solution, our primary focus is on the selected items and their cumulative value. To optimize the process, we modify the data structure by assuming that each cell contains a pair in the form of `(value: int, item: str)`. This adjustment eliminates the need for an additional step of backtracking through the table to identify the selected items, streamlining the overall process.

Code

```python
class DynamicProgramming:
    def camping_with_knapsack(self, items,total_capacity, number_of_items):
        dp = [[(0,'')]*(total_capacity+1) for _ in range(number_of_items+1)]
        for row in range(1, number_of_items+1):
            for col in range(1, total_capacity+1):
                if items[row-1]['weight']<=col:
                    candidate1 = dp[row-1][col]
                    candidate2 = (dp[row-1][col-items[row-1]['weight']][0]+items[row-1]['value'], (dp[row-1]
[col-items[row-1]['weight']][1]+", "+items[row-1]['name']).strip(',')) # (total_value, items_taken)
                    dp[row][col]=max(candidate1,candidate2)
                else:
                    dp[row][col] = dp[row-1][col]
        return dp

if __name__ == '__main__':
    items = [
    { 'name': 'Water', 'weight': 3, 'value': 10 },
    { 'name': 'Book', 'weight': 1, 'value': 3 },
    { 'name': 'Food', 'weight': 2, 'value': 9},
    { 'name': 'Jacket', 'weight': 2, 'value': 5},
    { 'name': 'Camera', 'weight': 1, 'value': 6}
    ]
    total_capacity = 6
    number_of_items = len(items)
    dp = DynamicProgramming()
    result = dp.camping_with_knapsack(items, total_capacity, number_of_items)
    total_value, items = result[number_of_items][total_capacity]
    print("Optimal set of items to take: ",items)
    print(f"Total value: ${total_value}")
```

Next Page

Output

```
#the cells in the matrix contain value, item(s) pair
dp=[
        [(0, ''), (0, ''), (0, ''), (0, ''), (0, ''), (0, ''), (0, '')],
        [(0, ''), (0, ''), (0, ''), (0, ''), (0, ''), (0, ''), (0, '')],
        [(0, ''), (0, ''), (0, ''), (0, ''), (0, ''), (0, ''), (0, '')],
        [(0, ''), (0, ''), (0, ''), (0, ''), (0, ''), (0, ''), (0, '')],
        [(0, ''), (0, ''), (0, ''), (0, ''), (0, ''), (0, ''), (0, '')],
        [(0, ''), (0, ''), (0, ''), (0, ''), (0, ''), (0, ''), (0, '')]
        ]

# after populating the matrix we get
dp =
[(0, ''), (0, ''), (0, ''), (0, ''), (0, ''), (0, ''), (0, '')]
[(0,''),(0, ''),(0, ''),(10,' Water'),(10,'Water'),(10,'Water'),(10,'Water')]
[(0,''),(3,' Book'),(3,'Book'),(10,'Water'),(13,'Water,Book'),(13,'Water,Book'),(13, ' Water,Book')]
[(0, ''),(3,'Book'),(9,'Food'),(12,'Book,Food'),(13,'Water,Book'),(19,' Water,Food'),(22,'Water,Book,Food')]
[(0,''),(3,'Book'),(9,'Food'),(12,'Book,Food'),(14,'Food,Jacket'),(19,' Water,Food'),(22,'Water,Book,Food')]
[(0,''),(6,'Camera'),(9,'Food'),(15,'Food,Camera'),(18,'Book,Food,Camera'), (20,'Food,Jacket,Camera'),
(25,'Water,Food,Camera')]
#returns the populated table

#prints out
Optimal set of items to take:   Water, Food, Camera
Total value: $25
```