Research on solving 0-1 knapsack problem by dynamic programmin

Zitong Gao

Yuhang High School, Hangzhou, Zhejiang Province, China, 310000

a7phalt@gmial.com

Abstract. The 0-1 Knapsack Problem poses the challenge of maximizing the total value of a subset of items, while respecting a capacity constraint in a knapsack. This paper explores the problem's solution and aims to find an optimal solution. The Dynamic Programming approach, based on the principles of optimal substructure and overlapping subproblems, breaks down the problem into smaller, solvable subproblems. The article discusses the key features of Dynamic Programming, including its ability to handle non-linearities and provide a bottom-up approach for optimal substructure. A code implementation showcases the pseudocode, demonstrating the iterative calculation and efficient storage of results. Reviewing the research, the Dynamic Programming is a suitable and powerful tool as the benefits are summarized and listed into less computations, high efficiency and the optimal final result. By leveraging Dynamic Programming, the 0-1 Knapsack Problem can be solved optimally by finding the subset of items that maximizes the value within the capacity constraint.

Keywords: 0-1 knapsack problem, dynamic programming, bottom-up approach, memorization.

1. Introduction

0-1 Knapsack Problem is where we are required to fill the knapsack of capacity C with a given subset of n items. Each item i has its certain weight W_i and value V_i. The Problem is to maximize the total value of all the items in the knapsack without exceeding the capacity C, while an item can only be packed one time and can't be divided. So, we can write a integer linear programming model:

$$\text{maximum value } f(x) = \sum_{i=1}^{n}(V_i x_i) \tag{1}$$

$$\text{subject to } \sum_{i=1}^{n} W_i x_i \leq C \tag{2}$$

$$x_i \in \{0,1\}, i \in \{1,2,\dots,n\} \tag{3}$$

where the variable x_i represents whether the item i is packed, x_i=1 if the item i is in the knapsack [1]. Without loss of generality, all coefficients are supposed to be positive integers. To avoid trivial cases we set that $W_i < C$ for every $i = 1,\dots,n$.

In this paper, we apply Dynamic Programming to find a general solution to this problem and build an easily achieved way to find an answer in computer language. This paper explains the Knapsack Problem and Dynamic Programming, gives a piece of code to solve 0-1 Knapsack Problem which can ensure to get the optimal solution.

2. Dynamic programming

The dynamic programming algorithm is a strong algorithm design technology first proposed by Bellman in the 1950s, that has been widely used in solving optimization problems in a multi-stage decision process. In this type of problem, the solutions are flexible and can be found where each of them corresponds to a value and there is a best solution with the optimal value [2].

Dynamic Programming is based on the principle of "optimal substructure," which states that an optimal solution to a larger problem can be constructed from optimal solutions to its smaller subproblems[2] The key idea of Dynamic Programming is to divide the problem into same smaller problems and try to solve them using the Bottom-Up approach, which suggests starting with the smallest subproblems and uses their solutions to solve larger and more complex subproblems. After that, examine whether each optimal decision-making sub-sequence contains the optimal decision making sub-sequence, that is, whether it has optimal sub-structural properties. To achieve higher efficiency and a more accurate result, we use a recursive formula to store the best solution of sub-problems and find out the answers when needed, so that it can avoid repetitive calculations to some extent and get a polynomial time algorithm. To be more specific, when we try to solve a problem with Dynamic Programming, the problem could first be divided into several steps, where each step only cares about the current best. For each step, there may be several related sub-problems, and Dynamic Programming requires selecting the optimal one and adding value to this step, which is a process of forming the optimal solution for this step.

So in total, Dynamic Programming has four main features:

First, overlapping subproblems, where the results of sub-problems are stored and then use to avoid redundant calculations.

Second, optimal substructures, where an optimal solution to a problem can be optimal constructed from optimal solutions of its substructures.

Third, memorization, where the results of expensive functions are stored and reused when the same inputs occur.

Fourth, bottom-Up approach, where the simplest subproblems are solved first and their solutions are used to solve more significant subproblems, to ensure that no computations are repeated, leading to improved efficiency and faster overall computation.

Overall, Dynamic Programming provides an efficient way to solve complex problems by breaking them down into smaller, more manageable subproblems and reusing the solutions to these subproblems to find the optimal solution to the original problem [3].

3. General outline of the algorithm

Connecting Dynamic Programming to the Knapsack Problem, this paper discuss its application in solving 0-1 Knapsack Problem.

When trying to maximize the total value of items in the knapsack, Dynamic Programming considers putting in the items one by one, storing all the possible results, and using them when putting in the next item. So the process can be described in steps:

First, considering the remaining capacity, put in one item. This item can be utilized by everyone as the capacity is greater than or equal to its weight. All the results, including the total value after putting in this item and the rest capacity, are stored corresponding to the input, the item putting in and the capacity enabled.

Second, update the capacity, and input the new available capacity and numbers of items.

Third, search in the stored results and consider whether the result of this input has already been calculated and stored. If yes, add the value in the result to the total value and update the capacity. If not, find all the possible results and store them.

Fourth, repeat step1-3 until we find the best final result [4].

To achieve this process, Dynamic Programming creates a two-dimensional array so the results can be distinguished and searched by combining the available capacity and number of items. So the two-dimensional array is firstly created as a matrix with a size of n*c, where n represents the number if items, and c is the capacity. Every line is initialized as [0] *c, number if column is i, and the number of items [5].

After the array is created and initialized, the items are put one by one and the results are stored.

For example, when there are 5 items for options where w= [1,3,4,5] and v= [8,10,15,20], c=5.

The initial array is shown below:

i\c	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0

Then we do the first round of calculation, putting in one item and finding the maximum value.

i\c	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	8	8	8	8	8
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0

Then keep on calculating, and when the item number is bigger than 1, we need to choose whether to put in the next item. If the next item is chosen to be put, the capacity is turned into the value after the original capacity subtracting the weight of the item. If not, the capacity will stay not change while the number of items subtracts 1.

In this example, when i=2 and c=5, we need to consider whether to put in the item 2 or not. It can be seen that c-w(1)=4, which is bigger than the weight of item 2, so item 2 is put in and the value of item 2 is added to the total value. After putting in item 2, available capacity turned into 1 and we need to consider whether to put in item 3 or not. However, the available capacity is smaller than w[3], so item 3 can't be put in.

With this routine calculation, we can get the final result as follow:

i\c	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	8	8	8	8	8
2	0	8	8	8	18	18
3	0	8	8	8	18	18
4	0	8	8	8	18	18

So in conclusion, the process of Dynamic Programming working in 0-1 Knapsack Problem is to calculate the results for all possible inputs, store the results, reuse them when doing the forward calculation if possible, ergodic all inputs, and get the final result.

4. Code implementation

To realize this idea in a computer and have an efficient way of doing this process, this paper provides a general pseudocode to explain how it works in computer.

First, to achieve the two-dimensional array, create the array "dp" to store the results, where dp[i][j] represents the result where the number of items is i and available capacity is j. It iterates through the items and the capacity to calculate the maximum value for each subproblem. Finally, it performs a backtrack to identify the selected items that contribute to the maximum value [6].

So the pseudocode could be as follows:

```
def function knapsack(weights, values, capacity):
    n = length(weights)
    dp = new two-dimensional array with dimensions (n + 1) x (capacity + 1)

    // Initialize the table with base cases
    for w = 0 to capacity:
        dp[0][w] = 0

// Dynamic Programming: Fill in the table
//first, Iterate over each item from 1 to n and for each weight from 0 to capacity
    for i = 1 to n:
        for w = 0 to capacity:
            if weights[i-1] <= w:
                dp[i][w] = max(values[i-1] + dp[i-1][w-weights[i-1]], dp[i-1][w])
            else:
                dp[i][w] = dp[i-1][w]
```

//Check if the weight of the current item (weights[i-1]) is less than or equal to the current weight (w) and decide whether to put in the item. If the weight of the current item is feasible to be included, calculate the maximum value that can be achieved by either including or excluding the current item. The maximum value is determined by comparing the sum of the current item's value and the value obtained from the remaining capacity (dp[i-1][w-weights[i-1]]) with the value obtained by excluding the current item (dp[i-1][w]). Assign the maximum value to dp[i][w].

If the weight of the current item is not feasible to be included, assign the value obtained from excluding the current item to dp[i][w].//

```
    // Backtrack to find the selected items
    selectedItems = []
    w = capacity
    for i = n to 1:
        if dp[i][w] != dp[i-1][w]:
            selectedItems.append(i-1)
            w = w - weights[i-1]
```

//Start from i = n (the last item) and the current weight w = capacity. If the value at dp[i][w] is different from the value at dp[i-1][w], it means the current item was included. Add its index (i-1) to the selectedItems list and subtract its weight from w.//

```
    return dp[n][capacity], selectedItems
```

In the above pseudocode, weights and values represent the arrays of weights and corresponding values of items, respectively. “capacity” represents the maximum weight the knapsack can hold. The function returns the maximum value that can be obtained and a list of indices for the selected items.

To test whether this pseudocode works well, rewrite it in the code version and do an experiment in Python. The code and result is shown below:

```python
def knapsack(weights, values, capacity) :
    n = len(weights)
    dp = [[0] * (capacity + 1) for _ in range (n + 1)]
    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights [i - 1] <= w:
                dp[i] [w] = max(values [i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w])
            else:
                dp[i] [w] = dp[i - 1] [w]

    selectedItems = []
    w = capacity
    for i in range(n, 0, -1):
        if dp[i] [w] != dp [i - 1][w]:
            selectedItems.append(i - 1)
            w -= weights [i - 1]
    return dp[n][capacity], selectedItems

w = [2,3,5,7]
v = [4,5,6,7]
capacity = 13

max_value, selected_ items = knapsack(w, v, capacity)

print ("Maximum Value:", max_value)
print ("Selected Items:", selected_items)
>>>
Maximum Value: 16
Selected Items: [3, 1, 0]
```

So it can be seen that this code can successfully achieve the answer.

5. Review the benefits

Compared to other solutions of 0-1 Knapsack Problem, Dynamic Programming is recommended for the reason that it performs well in both efficiency and result.

The superiority is achieved by these features:

First, avoidance of Redundant Computations: Dynamic Programming eliminates redundant computations by solving smaller subproblems first and storing their solutions. This way, when solving larger subproblems, the algorithm can reuse the already computed solutions, saving time and computational resources.

Second, improved Efficiency: By breaking down the problem into smaller subproblems and solving them independently, Dynamic Programming significantly improves efficiency. It reduces the time complexity from exponential (in the case of a brute-force approach) to polynomial, making it feasible to solve even larger instances of the Knapsack problem efficiently.

Third, optimal Solution: Dynamic Programming guarantees finding the optimal solution to the Knapsack problem. It systematically considers all possible combinations of items and weights, ensuring that the solution obtained is the one that maximizes the value within the given capacity.

Fourth, scalability: Dynamic Programming offers scalability, allowing the Knapsack problem to be solved efficiently for larger problem instances. The bottom-up approach, in particular, ensures that computations are performed iteratively, without any risk of stack overflow or excessive memory usage, enabling the algorithm to handle large inputs.

6. Conclusion

Through the analysis, we have explored the working principles of Dynamic Programming and its application in solving the 0-1 Knapsack problem. This research have witnessed how the step-by-step iterative process allows us to build a dynamic programming table, solve subproblems, and eventually derive the optimal solution for the original problem.

The benefits of using Dynamic Programming to solve the Knapsack problem are evident. It guarantees the optimal solution, overcomes subproblem dependencies, considers future choices, optimizes globally, and handles non-linearities. These advantages, coupled with the versatility of Dynamic Programming in addressing various optimization problems, make it a valuable tool for researchers and practitioners in computer science and related fields.

As we continue to explore and develop new algorithms and problem-solving techniques, Dynamic Programming remains a fundamental approach worth mastering. Its ability to break down complex problems, efficiently solve subproblems, and deliver optimal solutions has made it a cornerstone in algorithm design and optimization. With its wide range of applications, Dynamic Programming continues to shape and enhance our understanding of solving challenging computational problems.

In summary, Dynamic Programming provides a robust and reliable framework for solving the 0-1 Knapsack problem and offers valuable insights into the broader field of algorithmic problem-solving. By understanding its principles and techniques, we equip ourselves with a powerful tool to tackle a wide range of optimization problems efficiently and effectively.

References

[1] Martello, S. , & Toth, P. P. . (2011). Dynamic programming and strong bounds for the 0-1 knapsack problem. Management ence, 45(3), 414-424.
[2] Wang, Y. , Wang, M. , Li, J., & Xu, X.. (2020). Comparison of genetic algorithm and dynamic programming solving knapsack problem.
[3] Lan Wenfei, Wu Ziying, & Yang Bo. (2016). An improved dynamic programming algorithm for the knapsack problem. Journal of South-Central University for Nationalities: Natural Science Edition, 35(4), 5.
[4] Yan Li. (2020). Algorithmic Decision Analysis of 0-1 Knapsack Problem. Computer Knowledge and Technology: Academic Edition (4), 3.
[5] Sun Jianing, Ma Hailong, Zhang Lichen, & Li Peng. (2022). A Backtracking Algorithm for Solving the 0-1 Knapsack Problem Fusion Greedy Strategy. Computer Technology and Development (002), 032.
[6] Wang Maoping. (2021). Extended model of discounted {0-1} knapsack problem and its dynamic programming solution (Master's thesis, China West Normal University).