Solving Knapsack Problem using Dynamic Programming

Sriyani Violina
Iwa Ovyawan Herlistiono

DOI: https://doi.org/10.37178/ca-c.22.1.127

Sriyani Violina, *Informatics Department, Engineering Faculty, Widyatama University Jalan Cikutra No 204A Bandung*
Email: sriyani.violina@widyatama.ac.id

Iwa Ovyawan Herlistiono , *Informatics Department, Engineering Faculty, Widyatama University Jalan Cikutra No 204A Bandung*
Email: ovyawan.herlistiono@widyatama.ac.id

Abstract

Dynamic Programming is an algorithm or method in solving complex problems by describing how to solve them in several stages by dividing the problem into simpler problems. So that each stage of completion is related to each other until a final solution is produced. The Knapsack problem is a combinatorial problem, which is given a set of items, each of which has a weight and value. Knapsack solution using Dynamic Programming does not always get optimal results.

Keywords: Dynamic Programming, Discrete Knapsack

Introduction

Dynamic Programming is an algorithm or method in solving complex problems by describing how to solve them in several stages by dividing the problem into simpler problems. So that each stage of completion is related to each other until a final solution is produced. The idea of this dynamic program is very simple, if it has been solved with input in such a way, the solution is saved for future reference where if the same problem is found then there is no need to recompute. If a problem is given which is divided into several sub-problems and the sub-problems overlap with problems that have been solved previously, there is no need to search for solutions to these sub-problems. There are two ways in dynamic programming, namely Top-Down which is also known as memoization and Bottom-Up[1].

Dynamic Programming is commonly used in the fields of mathematics, bioinformatics, computer science, and economics. Not all problems can be solved by dynamic programs, problems that are usually solved by dynamic programs are optimization problems. Besides being used to find solutions to problems, dynamic programs are also commonly used to find the optimal number of solutions that can be generated, such as the problem of currency exchange and integer knapsack. Some examples of common problems that can be solved using dynamic programs are finding the shortest tour of a graph, Traveling Salesman Person (TSP) problems, capital budgeting problems (Capital Budgeting), integer knapsack, Sequence alignment, Fibonacci sequences, and various other problems.

The Knapsack Problem is an optimization problem that can be solved by several standard algorithms. The Knapsack problem is a combinatorial problem, where given a set of items each having a weight and value, the problem is how to choose which

items to include in the knapsack with the optimal total value and the total weight must be less than or equal to the size of the knapsack[2].

This paper will discuss about solving knapsack cases using Dynamic Programming.

Method

Problem Knapsack

In general there are two types of Knapsack problems. The first type is 0-1 Knapsack. The formal definition of 0-1 Knapsack is, suppose there are n items[3]:

$S = \{item_1, item_2, item_3, \ldots..item_n\}$

w_i = $item_i$ weight

p_i = $item_i$ profit

W = max capacity knapsack.

With wi, pi and W being positive integer numbers, then find the subset of A members of S such that:

$$\sum_{item\ i\ \in\ A} Pi \text{ maximum, } \sum_{item\ i\ \in\ A} wi\ \leq W.$$

In other words, the 0-1 Knapsack problem is how to take as many items as possible and put them in a knapsack with a capacity of W, with the total item weight must be less than or equal to W.

While the second type is Fractional Knapsack. In this type of Knapsack, we don't have to take the whole item i. Item i may be taken in parts.

Dynamic Programming

In dynamic programming, the solution characteristics are as follows[4]:

1. There are a finite number of possible options to choose from.
2. Solutions are completed in stages; Each stage of the solution is built from the results of the solution of the previous stage.
3. To limit the options that must be considered at each stage, optimization requirements and constraints are used.

The series of decisions made in a dynamic programming such that the optimal is determined by the optimality principle. Basically the principle of optimality reads as follows: "if the total solution is optimal, then the part of the solution up to the nth stage is also optimal". With the principle of optimality, then if we work from stage n to stage n + 1, we can use the optimal result at stage n without recomputing from the initial stage.

The steps for developing dynamic programming algorithms are as follows:

1. Establish a recursive property that gives the solution to an instance of the problem
2. Solve an instance of the problem in a bottom-up fashion by solving smaller instance first

Testing Data

The testing of these two algorithms will be carried out on one case sample, each consisting of 4 items with varying profits and weights as shown in table 1, with a knapsack capacity of 10.

Table 1

Data Test

Item	Profit	Weight	Density
1	50	2	25
2	40	5	8
3	60	10	6
4	15	3	5

Result and *Analysis*

Result

The first step is to determine the recurrence relation as follows:

$$V(i,j) = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ V(i-1,j) & \text{if } w_i>j \\ \max\{V(i-1,j), V(i-1,j-w_i)+ p_i\} & \text{if } w_i<=j \end{cases}$$

Then make a matrix with rows (i) 0-n (number of items), and columns are 0-knapsack capacity so that the matrix is formed as follows:

	0	1	2	3	4	5	6	7	8	9	10
0											
1											
2											
3											
4											

Iterate over the recurrence relation:

i=0

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1											
2											
3											
4											

i=1

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	50	50	50	50	50	50	50	50	50
2											
3											
4											

i=2

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	50	50	50	50	50	50	50	50	50
2	0	0	50	50	50	50	50	90	90	90	90
3											
4											

i=3

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	50	50	50	50	50	50	50	50	50
2	0	0	50	50	50	50	50	90	90	90	90
3	0	0	50	50	50	50	50	90	90	90	90
4											

i=4

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	50	50	50	50	50	50	50	50	50
2	0	0	50	50	50	50	50	90	90	90	90
3	0	0	50	50	50	50	50	90	90	90	90
4	0	0	50	50	50	65	65	90	90	90	105

Optimal results are achieved when the selected items are 1, 2 and 4 with a total profit of 105 and a total weight of 10.

Analysis

Computational repetition can be avoided by using temporary tables that hold the results or solutions that are solved in sub-problems in certain stages. So that in the next stage, the solution in the previous stage can be used by looking at the table that has been made. Thus, dynamic programs will run with faster execution times.

Knapsack problem solving with Dynamic Programming does not always reach the optimal solution.

Conclusion

Knapsack problem solving with Dynamic Programming does not always reach the optimal solution. Computational repetition can be avoided by using temporary tables that hold the results or solutions that are solved in sub-problems in certain stages. So that in the next stage, the solution in the previous stage can be used by looking at the table that has been made. Thus, dynamic programs will run with faster execution times

References

1. Amini, A.A., T.E. Weymouth, and R.C. Jain, *Using dynamic programming for solving variational problems in vision.* IEEE Transactions on pattern analysis and machine intelligence, 1990. **12**(9): p. 855-867.DOI: https://doi.org/10.1109/34.57681.
2. Hifi, M. and N. Otmani, *An algorithm for the disjunctively constrained knapsack problem.* International Journal of Operational Research, 2012. **13**(1): p. 22-43.DOI: https://doi.org/10.1504/IJOR.2012.044026.
3. Chu, P.C. and J.E. Beasley, *A genetic algorithm for the multidimensional knapsack problem.* Journal of heuristics, 1998. **4**(1): p. 63-86.DOI: https://doi.org/10.1023/A:1009642405419.
4. Bellman, R., *Dynamic programming*. Science, 1966. **153**(3731): p. 34-37.DOI: https://doi.org/10.1126/science.153.3731.34.