A Comparison of Greedy Algorithm and Dynamic Programming Algorithm

Xiaoxi Chen*

High School Affiliated to Renmin University of China, Beijing, China

ABSTRACT: Two algorithms to handle the problem include greedy algorithms and dynamic programming. Because of their simplicity, intuitiveness, and great efficiency in addressing problems, they are frequently employed in a variety of circumstances. The connection and difference of the two algorithms are compared by introducing the essential ideas of the two algorithms. The knapsack problem is a classic problem in computer science. In the application of solving the backpack problem, greedy algorithm is faster, but the resulting solution is not always optimal; dynamic programming results in an optimal solution, but the solving speed is slower. The research compares the application properties and application scope of the two strategies, with the greedy approach being the better approach in solving the knapsack problem.

1. INTRODUCTION

Computational algorithms have rapidly developed to satisfy people's need for large-scale data processing and the solution of a wide range of practical problems. Several models, including linear planning, dynamic programming, and greedy strategy, have been applied to computer computational law, resulting in efficient algorithms for solving a wide range of practical issues. In computational algorithms, dynamic programming algorithms and greedy algorithms are key core design principles. There are some commonalities as well as significant variances. The purpose of this research is to provide programmers with a practical application performance of both algorithms when choosing an optimized method to implement a function. With the above support, the programs using these two algorithms will have more decent data organization and clearer logic.

The optimal decision of a process has the property that its future strategies must constitute the optimal strategy for the process that takes the state established by the first decision as to its starting state, regardless of what its beginning state and initial decision are. In other words, an optimum strategy's sub-strategies must likewise be optimal for its beginning and final states. In general, fine-tuning the algorithm is required to attain higher performance. However, in some circumstances, even after adjusting the algorithm, the performance still falls short of the criteria, necessitating the use of another way to tackle the problem.

The partial knapsack problem and the 0/1 knapsack problem are discussed in this work, as well as the differences between greedy and dynamic programming algorithms. Practitioners in the field of computing are always faced with the selection between different algorithms in programming, and this paper helps them to choose the proper and efficient algorithm to complete their tasks in their programming work.

2. STATING THE KNAPSACK PROBLEM

In the knapsack problem, you are given n items (each item has just one item) and a knapsack. Item i has a weight of wi, a value of vi, and a capacity of C in the knapsack. Inquire about how to choose stuff so that the objects in the knapsack have the greatest value. For example, each item i load into xi has a benefit of vi *xi. There are two types of knapsack problems:

1. Partial knapsack problem. Items can be grouped into portions in a rucksack during the selecting process, i.e., $0 < xi < 1$ (greedy algorithm).
2. 0/1 knapsack problem. Similar to the partial knapsack issue, except with no load or full load, i.e. xi=1 or 0. (dynamic programming algorithm) [1].

3. GREEDY ALGORITHMS

The ideal solution is the row solution. A series of locally optimal choices, termed greedy choices, can lead to the global optimal solution to this type of problem (this is the main difference between greedy algorithms and dynamic programming).

3.1. Definition of greed strategy

A greedy strategy is a method of solving a problem from its initial state by making several greedy choices to arrive at the optimal value (or better solution). The greedy strategy always makes the choice that seems optimal at the moment, that is, the greedy strategy does not consider the

*Corresponding author. Email: 3408663616@qq.com

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (http://creativecommons.org/licenses/by/4.0/).

problem as a whole, but makes a choice that is only locally optimal in some sense, while the characteristics of many problems determine that the problem can be solved optimally or better using the greedy strategy. (Note: The greedy algorithm does not produce an overall optimal solution for all problems, but it does produce an overall optimal solution for a fairly wide range of problems. However, the solution is necessarily a good approximation to the optimal solution.) [2].

By using a top-down, iterative approach to make successive greedy choices, each greedy choice reduces the problem to a smaller sub-problem. To determine whether a specific problem has the property of greedy selection, we must prove that the greedy choices made at each step ultimately lead to an optimal solution to the problem. It is often possible to first show that an overall optimal solution to the problem starts with a greedy selection and that after the greedy selection is made, the original problem reduces to a similar sub-problem of smaller size. Then, it is shown by mathematical induction that each step of greedy choice leads to an overall optimal solution to the problem.

3.2. Practical application of greedy algorithms

3.2.1. The fundamental strategy of the greedy method

Starting from a certain initial solution to the problem. Approach the given goal step by step to find a better solution as fast as possible. When a certain step in the algorithm is reached and no further progress can be made, the algorithm stops.

3.2.2. Problems with the greedy algorithm

The final solution is not guaranteed to be the best.
It cannot be used to find the maximum or minimum solution.

It can only find the range of feasible solutions that satisfy certain constraints.

Table 1. Examples of knapsack problem

Items	A	B	C	D	E	F
Weight	10	30	40	20	10	20
Value	50	45	60	20	30	40
W/V	5	1.5	1.5	1	3	2

3.2.3. The process of implementing the algorithm

Starting from an initial solution of the problem; finding a solution element of a feasible solution when it is possible to go further towards the given overall goal; combining all solution elements into a feasible solution of the problem.

3.2.4. Example of a knapsack problem (partial knapsack problem)

We suppose now there are 6 items, n=6 and W =90. Table 1 gives the weight, value and value per unit weight of the items.

To get the optimal solution, the knapsack must be loaded to the maximum capacity, i.e., W =

To solve this problem with a greedy strategy, firstly choose a metric, i.e., what criteria to follow in each selection. After analysis, this problem can be in accordance with the criteria of maximum value, minimum weight, and maximum value per unit weight, respectively [3]. The analysis is as follows.

Metrics according to maximum value priority:

The first choice among the remaining optional items is the one with the highest value, i.e., item C, which weighs 40 and is smaller than the total capacity of the knapsack, and then items A and B. Consequently, it cannot be put in. The corresponding solution list is:

$$x = [1,1,1,0,0,0]$$

Metrics according to minimum wright priority:

Select the item with the least weight first from the remaining available items each time. Select in order. That is, first select item 1 with a weight of 10, which is smaller than the total capacity of the knapsack of 90, and then select items 5, 4, 6, and 2 in turn. The total capacity and value of the selected items are respectively:

$$\sum C = 10 + 10 + 20 + 20 + 30 = 90$$

$$\sum V = 50 + 30 + 20 + 40 + 45 = 185$$

The corresponding solution list is:

$$x = [1,1,0,1,1,1]$$

After comparison, the total value obtained by selecting the item according to the criterion of minimum weight is greater than the total value obtained by selecting the item according to the criterion of maximum value. That is, the weight criterion is superior to the value criterion.

Metrics according to maximum unit wright priority:

Each time in the remaining optional items first choose the item with the largest unit value, and then choose in turn. After analysis, the order of selecting items in turn is 1, 5, 6, 2, at this time the capacity of the knapsack is:

$$\sum C = 10 + 10 + 20 + 30 = 70$$

C is less than the total capacity of the knapsack 90, you can also put half of item C, the total weight of 90, at this time the total value of the knapsack is:

$$\sum V = 50 + 30 + 40 + 45 + 30 = 195$$

After comparing, this method is optimal.

Therefore, the selection strategy of the 0/1 knapsack problem is to select the items according to the maximum unit value first, and then greedily select the item with the largest unit value among the available items. To solve the problem, first, sort the n items by unit value, and then prioritize the items according to the largest unit value after sorting.

4. DYNAMIC PROGRAMMING

4.1. Principles of dynamic programming

The basic strategy of dynamic programming is the multi-stage problem, which is a problem that can be divided into multiple interconnected stages with a chain structure in a specific order. At each stage, a decision needs to be made, and the decision of the previous stage directly affects the state of the next stage, which depends on the result of the previous decision. The decisions of all stages will eventually form a decision sequence and solving a multi-stage decision optimization problem is to find a decision sequence that makes a certain indicator function of the problem optimal [5].

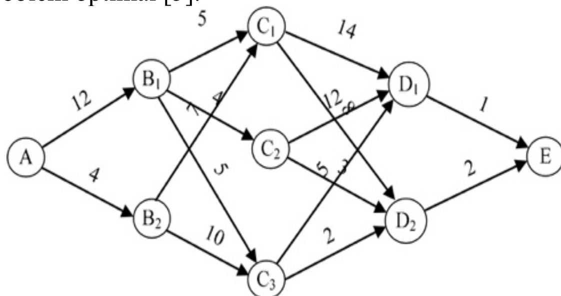

Figure 1. Multi-stage decision strategy.

As shown in Figure 1, solving problem A depends on solving several sub-problems of phase B, solving phase B depends on solving several problems of phase C, and so on until all problems are solved.

4.2. Scope of application of dynamic programming

The dynamic programming algorithm applies a certain range and prerequisite constraints, beyond which the specific certain conditions, it becomes useless. Decision optimization problems that can be solved by dynamic programming methods must satisfy the optimal substructure property (optimality principle) and the non-aftereffect nature of the state.

4.2.1. Optimal substructure properties

The first step in solving multi-stage decision problems with dynamic programming algorithms is often to describe the structure of the optimal solution to the problem. A problem is said to satisfy the optimal sub-structure property if the optimal solution to the problem is composed of optimal solutions to sub-problems. The optimal substructure property is the basis of dynamic programming algorithms. Any problem whose solution structure does not satisfy the optimal substructure property cannot be solved by dynamic programming methods. In general, the state transfer equation of the problem can be derived from the optimal substructure of the problem. In a dynamic programming algorithm, the optimal solution to the problem is composed of the optimal solutions to the sub-problems, so it is important to ensure that the sub-problems used to construct the optimal solution are solved during the runtime.

4.2.2. Non-aftereffect property

When a multi-stage decision problem is divided into stages, the state of the stages preceding a given stage will not influence the decision made in the current stage. The current stage can only make decisions about the future development of the process through the current state without depending on the state of the previous stages, which is called posteriority-free.

Therefore, the key condition for a problem to be solved by a dynamic programming algorithm is that the state of the problem satisfies the non-aftereffect property. To determine whether the states of the problem have non-aftereffect property, an effective method is to model the graph with the phase states as vertices and the decision relationships between phases as directed edges, and then determine whether the graph can be topologically ordered. If the graph cannot be topologically ordered, then there are loops and the problem is not non-aftereffect between the states, and the problem cannot be solved by dynamic programming.

4.3. Characteristics of dynamic programming problems

The effectiveness of the dynamic programming algorithm depends on an important property of the problem itself: the sub-problem overlap property.

When the algorithm computes the same sub-problem repeatedly during the run, it indicates that the sub-problems of the problem are overlapping. Dynamic programming takes advantage of the overlapping nature of sub-problems by computing each sub-problem encountered for the first time and caching the solution of the sub-problem so that if the algorithm encounters the same sub-problem in the next run, it does not need to recompute it and can directly check the cached results. The key to the dynamic programming algorithm is that it stores the various states of the solution during the process, avoiding the repeated computation of sub-problems. In contrast, the partitioning algorithm does not cache the solutions of sub-problems when solving a problem, and calculates a new sub-problem each time, so the sub-problems that can be solved by the partitioning algorithm cannot overlap, and if the sub-problems overlap, there will be a lot of repeated calculations in the partitioning algorithm, resulting in the inefficiency of the algorithm in time. The overlapping nature of sub-problems is not a necessary condition for applying dynamic programming algorithms, but the time efficiency of dynamic programming algorithms depends on the degree of overlapping sub-problems.

4.4. General steps of dynamic programming algorithm design

Define sub-problems: The problem is divided into several sub-problems based on the characteristics of the problem.

The divided sub-problems are sequentially related, and the current sub-problem can be solved given a relatively small sub-problem solution.

Selecting a state: The objective situation of the sub-problem is represented by the state, which must satisfy non-aftereffect.

Determining the state transfer equation: the process of determining the state of the current stage by choosing the state of the previous stage and the decision of the current stage is state transfer. Decisions are directly related to state shifting, and the state shifting equation for the problem can be written naturally if the range of decisions available for the stage is determined.

Finding the boundary conditions: the initial or end conditions of the iteration of the state transfer equation. There is no standard model of dynamic programming algorithm that can be used for all problems, and the algorithmic model of dynamic programming varies from problem to problem, so problem-specific analysis is needed. When designing an algorithm using dynamic programming ideas, it is not necessary to stick to the design model too much often have unexpected results.

4.5. Examples of Dynamic Programming Algorithms: 0/1 Knapsack Problem

When solving a real problem with a dynamic programming algorithm, the first step is to build a dynamic programming model, which generally requires the following steps:

Analyze the problem and define the characteristics of the optimal solution.

Divide the problem phase and define the phase calculation objectives.

Solve the phase conclusions, form a decision mechanism, and store the knowledge set.

Construct an optimal solution based on the information obtained when calculating the optimal value.

Design the program and write the corresponding code.

The optimal solution is to select n items ($0 \leq n \leq N$) so that V is maximum; the knapsack problem is an N-stage problem with j sub-problems in each stage, and the state is defined as the process of how to decide the state of C = j and N = i. The decision function is f i, j), and the analysis shows that f (i, j) The decision is shown in equation below, where vi is the value of V for the ith knapsack, which is the core algorithm of the decision:

$$f(i,j) = \begin{cases} \max\left(f(i-1, j-vi) + vi, f(i-1,j)\right) \\ f(i-1,j) \end{cases}$$

When $vi \leq j$, f (i, j) takes the maximum of f (i − 1, j − vi) + v(i) and f (i − 1, j); when vi > j, the ith knapsack cannot be put in, so the solution f (i, j) = f (i − 1, j).

In the equation, f (i − 1, j),f (i − 1, j − vi) are solved, so f (i, j) can be calculated [6].

5. COMPARISON OF DYNAMIC PROGRAMMING ALGORITHM AND GREEDY ALGORITHM

Both dynamic programming algorithms and greedy algorithms are recursive classical algorithms for solving optimization problems, and both derive the global optimal solution from the local optimal solution, which makes them similar. However, there are significant differences between them.

Each decision step of the greedy algorithm cannot be changed and becomes a definitive step in the final decision solution, shown in the equation below.

$$f(xn) = \bigcup_{i=0}^{n} Vi$$

The global optimal solution of the dynamic programming algorithm must contain some local optimal solution, but the optimal solution of the current state does not necessarily contain the local optimal solution of the previous state, so it is different from the greedy algorithm, which needs to calculate the optimal solution of each state (each step) and save it for the subsequent state calculation reference.

The greedy algorithm outperforms the dynamic planning algorithm in terms of time complexity and space complexity, but the "greedy" decision rule (decision basis) is difficult to determine, i.e., the selection of the Vi function, so that different decision bases may lead to different conclusions, affecting the generation of optimal solutions.

The dynamic programming algorithm can be used to solve the eligible problems in a limited time, but it requires a large amount of space because it needs to store the computed results temporarily. Although it is possible to share a single sub-problem solution for all problems containing the same sub-problem, the advantage of dynamic programming comes at the expense of space. The space conflict is highlighted by the need for efficient access to existing results and the fact that data cannot be easily compressed and stored. The high timeliness of dynamic programming is often reflected by large test data. Therefore, how to solve the space overflow problem without affecting the operation speed is a hot issue for dynamic programming in the future.

6. CONCLUSION

As with greedy algorithms, in dynamic planning, the solution to a problem can be viewed as the result of a series of decisions. as the result of a series of decisions. The difference is that in a greedy algorithm, an irrevocable decision is made every time the greedy criterion is used, whereas in dynamic programming, it is also examined whether each optimal sequence of decisions contains an optimal subsequence. When a problem has an optimal substructure, we think of using dynamic programming to solve it, but there are simpler, more efficient ways to solve some problems, if we always make what seems to be the best choice at the moment. The choices made by the greedy algorithm can depend on previous choices, but

never on future choices or on the solution of sub-problems, which gives the algorithm a speed advantage in both coding and execution. This gives the algorithm a speed advantage in both encoding and execution. If a problem can be solved by several methods simultaneously, the greedy algorithm should be one of the best choices. However, the greedy algorithm does not provide the overall optimal solution or the most desirable approximation for all problems, and its application area is much narrower than that of the backtracking method, so it is important to judge the right time for its application.

This paper has achieved some results in the time-efficient optimization of dynamic programming algorithm and greedy algorithm, but there is a need for further improvement and perfection. The next phase of research will focus on combining the dynamic programming algorithm and the greedy algorithm with other optimization algorithms respectively, to design high-performance programming models that bring into play the advantages of both and enhance the computational efficiency of the algorithms so that they can adapt to larger-scale computations. This is also the direction of improvement and development of these two strategies.

REFERENCES

1. Chu, P., & Beasley, J. (1998). A Genetic Algorithm for the Multidimensional Knapsack Problem. Journal of Heuristics, 4(1), 63–86. https://doi.org/10.1023/a:1009642405419
2. Martello, S., & Toth, P. (1987). Algorithms for Knapsack Problems. North-Holland Mathematics Studies, 132, 213–257. https://doi.org/10.1016/S0304-0208(08)73237-7
3. Vince, A. (2001). A framework for the greedy algorithm. DISCRETE APPLIED MATHMATICS. https://doi.org/10.1016/S0166-218X(01)00362-6
4. Wolsey, L. A. (1982). An analysis of the greedy algorithm for the submodular set covering problem. Combinatorica, 2(4), 385–393. https://doi.org/10.1007/bf02579435
5. Eddy, S. R. (2004). What is dynamic programming? Nature Biotechnology, 22(7), 909–910. https://doi.org/10.1038/nbt0704-909
6. Rahwan, T., & r Jennings, N. (2008). An Improved Dynamic Programming Algorithm for Coalition Structure Generation. https://aamas.csc.liv.ac.uk/Proceedings/aamas08/proceedings/pdf/paper/AAMAS08_0192.pdf