Proceedings of the 2023 International Conference on Machine Learning and Automation
DOI: 10.54254/2755-2721/35/20230392

A survey of dynamic programming algorithms

Yunong Zhang

Civil Engineering College, Xi'an University of Architecture and Technology, Xi'an, Shaanxi Province, China, 710054

zhang.yunong.0209@gmail.com

Abstract. Dynamic programming is an important algorithmic idea with widespread applications in computer science and other disciplines. With the continuous improvement of computing power and the increasing complexity of practical problems, dynamic programming algorithms have also received more and more attention. However, due to the limitations of the dynamic programming algorithm itself, it also brings challenges to the optimization of the algorithm and its application in different fields. This paper uses the methods of literature review and case analysis to systematically summarize and sort out the basic principles, complexity analysis, classic problems, and applications of dynamic programming algorithms. First of all, this paper discusses the basic concepts of splitting and optimal substructure characteristics of related problems in dynamic programming; secondly, it discusses the time complexity and space complexity of dynamic programming algorithms in detail; then, in the classic dynamic programming The case part focuses on the knapsack problem, including the 0-1 knapsack problem and multiple knapsack problems; finally, this paper discusses the wide application and optimization of dynamic programming algorithms in practice, covering natural language processing, bioinformatics and How to optimize the dynamic programming algorithm to improve the efficiency of problem solving and reduce resource consumption. To sum up, this paper fully demonstrates the basic principles and applications of dynamic programming algorithms, as well as optimization methods and development trends, and provides guidance and references for their academic and practical applications.

Keywords: algorithm, dynamic programming algorithm, knapsack problem.

1. Introduction

The dynamic programming algorithm is an important and widely used algorithm idea that shows powerful ability and flexibility in solving optimization problems. As a bottom-up solution method, the dynamic programming algorithm effectively solves many complex problems in practical applications by decomposing complex problems into smaller sub-problems and utilizing the properties of optimal substructures.

In the field of computer science, dynamic programming algorithms are widely used in areas such as algorithm design, optimization, sequence alignment, and path search. Its advantage is that many seemingly inefficient problems can be solved in polynomial time. By ingeniously designing the state definition and state transition equations of the problem, the dynamic programming algorithm can find the optimal solution or near-optimal solution in an efficient manner, thus providing important support for practical applications.

DOI: 10.54254/2755-2721/35/20230392

This article will comprehensively introduce the basic principles of dynamic programming algorithms, review complexity and classic case analysis, and discuss some extensions and applications of dynamic programming algorithms. By integrating the research results of dynamic programming algorithms on different issues and the current research frontiers and hotspots of dynamic programming algorithms, it provides researchers with a conceptual framework and methodology, and at the same time provides directions and inspiration for subsequent research. Dynamic programming algorithms will continue to play an important role in research and applications in computer science and other fields as a powerful tool.

2. Basic principles and complexity analysis of dynamic programming

2.1. Basic concept

Dynamic programming algorithm is a method to solve optimization problems by decomposing complex problems into smaller sub-problems and exploiting the optimal substructure properties [1]. The basic principle is to divide the original problem into a series of overlapping sub-problems, and solve these sub-problems in a recursive manner. Storing the solutions of the solved subproblems avoids repeated calculations, thereby improving the efficiency of the algorithm.

The core idea of a dynamic programming algorithm mainly includes three aspects: optimal substructure, recurrence relation, and overlapping subproblems [2]. The first is the optimal substructure property, that is, the optimal solution of a problem can be derived from the optimal solution of its sub-problems. Second, the recurrence relation is an important step in the solution of dynamic programming problems, because it describes the relationship between the current problem and its sub-problems. By defining state and state transition equations, a large problem can be decomposed into small problems, and the solutions of the subproblems can be used to find the optimal solution of the overall problem. Finally, there are overlapping sub-problems. The dynamic programming algorithm avoids repeated calculations by solving overlapping sub-problems again and again. In the process of solving the problem, the solutions to many sub-problems will continue to be used in the next calculation. By saving the solutions of the solved sub-problems, repeated calculations in recursion can be avoided and efficiency can be improved.

2.2. Time complexity

Time complexity is used to measure the degree to which the execution time of the algorithm increases with the input scale. It expresses the relationship between the time required for algorithm execution and the input scale, and is usually represented by the symbol (O).

The time complexity of the dynamic programming algorithm depends on the number of subproblems and the solution time of each subproblem. In general, the time complexity of the dynamic programming algorithm can be expressed as O(n*m), where n represents the size of the problem and m represents the number of states. It can be seen from this that as the scale of the problem increases, the time complexity of the dynamic programming algorithm increases polynomially, and in some cases it will increase exponentially, which is also a defect in solving large-scale problems. Therefore, some improved algorithms for time complexity have also been proposed, such as pruning strategies and approximation algorithms.

2.3. Space complexity

Space complexity is used to measure the degree to which the additional space or memory required by the algorithm grows with the increase of the input scale, and it represents the relationship between the additional space required for algorithm execution and the input scale.

The space complexity of the dynamic programming algorithm mainly depends on the size of the state table or matrix. Usually, the space complexity of the dynamic programming algorithm is also O(n*m), where n represents the size of the problem, and m represents the number of states, which means that a two-dimensional table or matrix needs to be created to store the solution of the sub-problem. In some

DOI: 10.54254/2755-2721/35/20230392

cases, space complexity can be reduced through optimization strategies, such as state compression techniques or saving only necessary subproblem solutions to reduce memory usage.

3. Classic dynamic programming problem: knapsack problem

3.1. Problem description

The Knapsack Problem is a classic combinatorial optimization problem, usually used to describe how to select a group of items to maximize the total value or minimize the total weight under a given knapsack capacity [3]. Specifically, the knapsack problem includes the following elements:

1. Backpack capacity: Given a fixed backpack capacity, usually represented by a positive integer, it represents the maximum weight or volume that the backpack can hold.

2. Collection of items: There is a set of items, each with its own value and weight. Value can indicate the importance or utility of the item, while weight indicates the backpack capacity the item occupies.

3. Selection restrictions: There may be certain restrictions on the selection of each item, for example, each item can only be selected once (0-1 knapsack problem) or can be selected multiple times (infinite knapsack problem).

The goal of the knapsack problem is to select a set of items that maximize their total value or minimize their total weight while satisfying the knapsack capacity constraints. Common backpack problems include:

0-1 Knapsack Problem: Each item can only be selected once;

Multiple Knapsack Problem: Each item has multiple optional copies, and each copy has a limit.

Unbounded Knapsack Problem: Each item can be selected multiple times.

3.2. Solution strategy

The general strategy of the dynamic programming algorithm to solve the knapsack problem is to record the optimal solution of the sub-problem through the dynamic programming table, and gradually solve the whole problem by filling in the table [4].

The first is to transform the original problem into a sub-problem, defining the rows and columns of the dynamic programming table as optional items and knapsack capacities, respectively. For example, assuming there are n items and a knapsack with a capacity of W, the rows of the dynamic programming table dp represent the indexes of the items, and the columns represent the capacity of the knapsack. That is, [i] represents the item, and [j] represents the capacity of the backpack.

Second, initialize the form. Initialize the boundary conditions of the dynamic programming table, namely the first row and first column. The boundary condition means that when there are no items to choose from or the capacity of the backpack is 0, the value in the backpack is 0.

Next, according to the characteristics and constraints of the problem, the state transition equation is designed to determine the relationship between the optimal solution at the current position and the previous position, and the state transition can be performed by comparing the benefits of different options.

If the i-th item can be put in the backpack, consider the two options of putting the item in the backpack or not putting it in the backpack, and choose the larger value as the optimal solution. That is,

$$dp[i][j] = max\big(dp[i-1][j], dp[i-1]\big[j - w[i] + v[i]\big]\big)$$

where w[i] represents the i-th The weight of the item, v[i] represents the value of the i-th item.

dp[i-1][j] indicates the current maximum value when the item is not put in; dp[i-1][j-w[i]] indicates that the first i-1 items correspond to when the backpack capacity is j-w[i] The maximum value of , that is, the value that it already has when it can just be put into this item. Then add the value v[i] of the item itself, which is the value of the item after it is put in.

If the i-th item cannot be placed in the backpack, then the optimal solution is the same as the optimal solution in the previous position. That is,

$$dp[i][j] = dp[i-1][j]$$

DOI: 10.54254/2755-2721/35/20230392

Finally, according to the state transition equation, traverse and fill the dynamic programming table dp. Starting at the upper left corner, calculate row by row or column by column until the entire table is filled. The final optimal solution to the knapsack problem is stored in the cell dp[n][W] in the lower right corner of dp, where n represents the number of items and W represents the capacity of the knapsack. By backtracking the dynamic programming table, starting from the lower right corner, according to the conditions of the state transition equation, the specific items and total value put into the backpack can be determined.

3.3. Examples and applications

(1)0-1 Knapsack Problem

Suppose there is a knapsack with a capacity C=10. The following items are now available to choose from in your backpack:

Item 1: weight w1=2, value v1=6

Item 2: weight w2=3, value v2=8

Item 3: weight w3=4, value v3=10

Item 4: weight w4=5, value v4=12

The requirement of the problem is to select items to maximize the total value of the backpack without exceeding the capacity its backpack.

For such problems, dynamic programming algorithms can be used to solve them.

First, a two-dimensional array dp can be used to represent the dynamic programming table. dp[i][j] represents the maximum value that can be obtained by putting the first i items into a knapsack with a capacity of j. And to initialize the table, initialize the first row and the first column of dp to 0, indicating that the maximum value is 0 when there is no item or the backpack capacity is 0.

Second, calculate each position of the dp table one by one according to the state transition equation:

If the i-th item can be put into the backpack (that is, the current backpack capacity is greater than or equal to the weight of the i-th item), consider putting the item in the backpack or not putting it in the backpack. Choose the larger value as the optimal solution.

If you choose to put it in the backpack:

$$dp[i][j] = dp[i-1]\left[j - w[i]\right] + v[i]$$

If you choose not to put it in the backpack:

$$dp[i][j] = dp[i-1][j]$$

If the i-th item cannot be placed in the backpack, then the optimal solution is the same as the optimal solution in the previous position:

$$dp[i][j] = dp[i-1][j]$$

The next step is to fill in the table and calculate: start from the upper left corner, calculate by row or column, and fill the entire dp table. Finally, the final maximum value is stored in the lower right cell dp[4][10] of the dp table, which is the last element of the dp table.

In this example, the dp table after calculation is shown in Table 1:

DOI: 10.54254/2755-2721/35/20230392

Table 1. 0-1 Knapsack problem dp table.

goods backpack capacity	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	6	6	6	6	6	6	6	6	6
2	0	0	6	8	8	14	14	14	14	14	14
3	0	0	6	8	10	14	16	18	18	24	24
4	0	0	6	8	10	14	16	18	20	24	26

Therefore, the maximum item value in the final backpack is 26. We can backtrack the dynamic programming table, starting from the lower right corner, and determine the specific items and total value put into the backpack according to the conditions of the state transition equation. In this example, the items selected to put in the backpack are items 1, 2, and 4, and their total value is 26.

(2)Multiple Knapsack Problem

Suppose there is a knapsack with capacity C=6. The following items are now available to choose from in the backpack:

Item 1: weight w1=2, value v1=4, available quantity n1=2

Item 2: weight w2=3, value v2=5, available quantity n2=3

Item 3: weight w3=4, value v3=6, available quantity n3=1

The requirement of the problem is to select items to maximize the total value of the backpack without exceeding its capacity.

Similar to the 0-1 knapsack problem, use the dynamic programming algorithm to create a two-dimensional array dp, where dp[i][j] represents the maximum value when considering the first i items and the knapsack capacity is j.

First, initialize the dp array to 0, and dp[i][j] represents the maximum value when no item is selected.

Secondly, considering that the number of items that can be used for each item is different for the multiple knapsack problem, it is necessary to use the nested loop method to traverse the items and knapsack capacity, and update the value in the dp array:

For each item i, traverse the knapsack capacity j (from 0 to C):

If people choose not to put items, then dp[i][j] can be updated to dp[i-1][j], which is to maintain the maximum value when putting i-1 items.

If people choose to put items in, then it is divided into two steps:

For each item i, we can choose to put k items i (0 <= k <= min(n[i], j/w[i])).

For each k, update dp[i][j] to dp[i-1][j-k*w[i]] + k*v[i], choose the largest value as the result of dp[i][j] .

Table 2. Multiple knapsack problem dp table.

goods backpack capacity	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	4	4	8	8	8
2	0	0	4	5	8	9	10
3	0	0	4	5	8	9	10

DOI: 10.54254/2755-2721/35/20230392

Therefore, the maximum item value in the final backpack is 12. This paper can backtrack the dynamic programming table, starting from the lower right corner, and determine the specific items and total value put into the backpack according to the conditions of the state transition equation. In this example, the items selected to be put in the backpack are 2 item 1 and 1 item 3.

4. Application and optimization of dynamic programming algorithms

4.1. Application

Dynamic programming algorithms have a wide range of applications in many fields, such as stochastic decision-making, natural language processing, and bioinformatics, for processing data such as text, speech, DNA sequences, and protein sequences.

In natural language processing, dynamic programming algorithms are widely used in tasks such as speech recognition, syntax analysis, machine translation, and text generation. By defining appropriate states and state transition equations, dynamic programming algorithms can process speech signals, text data, and find the best language model, syntax tree, or translation scheme. For example, in speech recognition, dynamic programming algorithms can solve problems such as feature extraction of audio signals, acoustic model matching, and vocabulary probability calculation, so as to achieve high-precision speech-to-text conversion.

In the field of bioinformatics, dynamic programming algorithms play an important role in DNA sequence alignment, protein structure prediction and genomics research. For example, in terms of protein structure prediction, AlphaFold has achieved great success in recent years. It is mainly based on deep learning and neural network technology, and predicts its three-dimensional structure by analyzing the amino acid sequence of proteins [5]. In protein structure prediction, problems such as sequence alignment and the longest common subsequence can be solved by dynamic programming algorithms; these problems can help identify conserved regions and similarities in protein sequences, thereby inferring their structure and function. AlphaFold uses a similar idea to infer the three-dimensional structure of the protein by analyzing and modeling the protein sequence. This has important implications for understanding the mechanisms of biological systems, disease research, and drug design.

Therefore, the application of dynamic programming algorithms in the fields of natural language processing and bioinformatics provides researchers with powerful tools and methods to solve complex problems. Through reasonable modeling of problems, state definition and design of state transition equations, dynamic programming algorithms can efficiently process and analyze large-scale data, bringing new progress and innovation to the research and application of these fields.

4.2. Optimization

The dynamic programming algorithm has certain limitations in terms of time and space complexity, but there are already many different optimization methods for dynamic programming algorithms for different problems. This study will briefly introduce three important optimization techniques: memory search, state compression dynamic programming, and multi-stage dynamic programming.

Memorized search is an optimization method of a dynamic programming algorithm. By saving the solutions of sub-problems that have been calculated, it avoids repeated calculations and improves the efficiency of the algorithm [6]. Memorized search can be regarded as top-down dynamic programming. Although it is also solved recursively, it uses a cache to store the calculated results when recursively solving the problem. When it needs to be calculated again, it directly gets it from the cache to avoid double calculations. Memorized search can significantly reduce the amount of computation and improve the efficiency of dynamic programming algorithms.

State compression dynamic programming is an optimization technique for problems with high-dimensional state spaces. In the traditional dynamic programming algorithm, the state is usually stored in a complete form, which requires a large space complexity. The state compression dynamic programming reduces the size of the state space and reduces the space complexity by compressing the high-dimensional state into a one-dimensional or low-dimensional state. State compression dynamic

DOI: 10.54254/2755-2721/35/20230392

programming improves the efficiency of the algorithm by ingeniously designing the state compression method and using bit operations or other techniques for state transfer and calculation while maintaining the correctness of the algorithm.

Multi-stage dynamic programming is a dynamic programming method applied to multi-stage decision-making problems. It decomposes the multi-stage decision-making problem into a series of sub-problems, and obtains the optimal solution by solving the sub-problems stage by stage. Multi-stage dynamic programming is similar to traditional dynamic programming, but it pays more attention to the characteristics and mutual influence of decision-making stages when solving problems. Through multi-stage dynamic programming, a complex multi-stage decision-making problem can be decomposed into a series of simple sub-problems, and the optimal solution can be obtained through state transition and step-by-step decision-making.

In summary, memorized search, state compression dynamic programming and multi-stage dynamic programming can all be regarded as improvements and extensions to the dynamic programming algorithm. By introducing new ideas and techniques, the efficiency and application range of the algorithm are improved.

5. Conclusion

The dynamic programming algorithm is a powerful and widely used algorithm idea that plays an important role in solving optimization problems. By dividing the problem into subproblems and exploiting the properties of optimal substructure and overlapping subproblems, dynamic programming algorithms can efficiently solve many complex problems. In this research, the basic principles of the dynamic programming algorithm are deeply discussed, including problem modeling, state definition, derivation and solution of the state transition equation. And through the analysis of classic cases such as the knapsack problem, it demonstrates the application of dynamic programming algorithms to practical problems. In addition, the paper also introduces some extensions and application areas of dynamic programming algorithms, such as memory search, state compression dynamic programming and multi-stage dynamic programming. These extensions and applications further enrich the application range of dynamic programming algorithms and provide more methods for solving complex problems.

Although the dynamic programming algorithm has certain limitations in terms of time and space complexity, its effectiveness and flexibility in solving optimization problems make it an important tool in research and practical applications. With the continuous improvement of computing power and the continuous advancement of algorithm improvement, dynamic programming algorithms can also be expected to play a greater role in a wider range of fields and problems in the future.

To sum up, the dynamic programming algorithm is a powerful algorithmic idea that has a wide range of applications in the solution of optimization problems. By deeply understanding and studying the principles and applications of dynamic programming algorithms, we can provide efficient solutions to practical problems and open up new possibilities for further research in computer science and other fields.

References

[1] Cormen T. Introduction to algorithms, thomas h. cormen, charles e. leiserson, ronald l. rivest, clifford stein[J]. Journal of the Operational Research Society, 2001, 42.

[2] Bertsekas D. Dynamic programming and optimal control: Volume I[M]. Athena scientific, 2012.

[3] Martello S, Toth P. Knapsack problems: algorithms and computer implementations[M]. John Wiley & Sons, Inc., 1990.

[4] Kellerer H, Pferschy U, Pisinger D, et al. Multidimensional knapsack problems[M]. Springer Berlin Heidelberg, 2004.

[5] Senior A W, Evans R, Jumper J, et al. Improved protein structure prediction using potentials from deep learning[J]. Nature, 2020, 577(7792): 706-710.

[6] Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms[M]. Society for Industrial and Applied Mathematics, 2010.