Proceedings of the 6th International Conference on Computing and Data Science
DOI: 10.54254/2755-2721/71/20241687

Representative applications of dynamic-programming in different fields: Analysis on problems of shortest-path, 0/1 knapsack and longest common sub-sequence

Ziyang Chia Zheng

Brooklyn College, Brooklyn, NY, 11210, USA

samyoungwu@gmail.com

Abstract. This article mainly focuses on exploring the current application status, concrete things implemented by dynamic programming and advantages of dynamic programming in problems (the concept and operation of dynamic programming and the role it plays in programs) in different fields, future development trends and areas for improvement. It also introduces the gaps and problems currently encountered by dynamic programming, then provides corresponding solutions to make it more helpful and efficient in the future. In this regard, by consulting different references and online resources written by previous scientists, it finds out that they all have different points of view on dynamic programming and different solutions to its shortcomings. In the current research, dynamic programming is the most widely used algorithm with few limitations that can achieve the global optimal solution at any time; however, it does not mean that it can achieve the local optimal solution at any time, which also indirectly results in the memory space it requires being larger than other optimization algorithms. After analyzing the literature and researching and testing multiple applications, it is obvious that dynamic programming can make the code run easier and faster than traditional methods, even though it will take up more space.

Keywords: Dynamic programming, Applications, shortest path, 0/1 knapsack, LSC.

1. Introduction

With the continuous enhancement and widespread application of contemporary computer algorithm knowledge, people increasingly rely on algorithms to help them solve complex problems in a short time. This has also led to the creation and use of more and more new and convenient algorithms, dynamic programming being a particular example. At present, dynamic programming is not only faster but also much simpler than traditional algorithms and coding ideas, which is not only a mathematical problem, but also contains more research on dynamic programming. However, the current research on it is not perfect and complete enough, and it still faces different challenges, such as how to extract the essence and remove the dross.

This paper deeply explores various application issues and the application status of dynamic programming. In addition, it also illustrates and discusses how dynamic programming can simplify and concrete the code in the program; for example: its specific concepts and operation methods are different from ordinary code; what kind of effects and results it will bring and so on. Then further discusses its current advantages, which are high efficiency and speed; the gaps and deficiencies in current research

DOI: 10.54254/2755-2721/71/20241687

are definitely one of its fatal shortcomings: space is exchanged for time. This shortcoming makes the speed faster and the memory consumption will be relatively large simultaneously, which will lead to problems when processing large amounts of data. In the future, the development trends should be: depending on how to improve it; the ideal would be to ensure minimal memory consumption while maintaining speed and efficiency. If this conjecture is realized and comes true, it will be a huge progress in the computer science field.

2. Concept and applications of dynamic programming

2.1. Concept of dynamic programming

Richard Bellman proposed dynamic programming in his book Dynamic Programming in 1957. However, programming does not represent its superficial meaning but represents a processing method. Dynamic programming is often used in conjunction with the divide-and-conquer idea, then solving the redundancy, so that the result of dynamic programming can always reach the optimal solution. The problem-solving method of dynamic programming is bottom-up, as Parallel and (Nearly) Work-Efficient Dynamic Programming, A DP algorithm solves an optimization problem by breaking it down to sub-problems, memorizing the answers to the subproblems, and using them to find the answer to the original problem. The subproblems are usually indexed by integers, referred to as states. With clear context, the study directly uses i to refer to "state i" by Xiangyun Ding, Yan Gu, and Yihan Sun mentioned [1]. The value of any i+1 stage is only related to the value of its previous stage i, and not related to the previous choice before i. By analogy, dynamic programming can find the optimal solution of the current state (note that the optimal solution of the current state is not a local optimum). But the disadvantage is that it wastes a lot of space and cannot reach the local optimal solution, which is a big problem waiting to be solved.

There are some different and representative applications of dynamic programming below, which will be helpful to illustrate and prove how essential and widely-used dynamic programming is.

2.2. Different applications of dynamic programming

2.2.1. The shortest path problem. As the name suggests, it is to find the shortest distance between two points. For example, there are three points A, B, and C; to find the shortest distance from A to C, people only need to add the shortest distances from A to B and B to C. The advantage of Bellman-Ford algorithm is that it can processing negative weight edges and detecting negative weight loops, and its time complexity (TC) is:

$$TC = O(VE) \tag{1}$$

V is number of vertices and E is the number of edges.

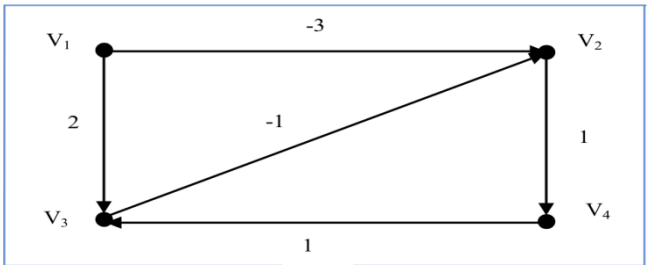

Figure 1. Bellman-Ford algorithm example [2].

All routes between two points shown in Figure 1 are based on the Bellman-Ford algorithm, the shortest path will be found by adding two by two and comparing all of them.

DOI: 10.54254/2755-2721/71/20241687

2.2.2. The 0/1 knapsack problem. The 0/1 kanpsack problem is required to choose to put items of corresponding value into the backpack or not into the backpack so that their total value is maximized, and the total weight cannot exceed the backpack capacity. Its time complexity is:

$$TC = O(nW) \tag{2}$$

n is the number of items, and W is the backpack capacity.

Table 1. Details of example knapsack problem item [3]

Items	A	B	C	D
Values	25	35	40	55
Weights	8	6	7	9

Table 2. Possible subsets of items [3]

A	B	C	D	Total weight	Total value
0	0	0	0	0	0
0	0	0	1	9	55
0	0	1	0	7	40
0	0	1	1	16	95
0	1	0	0	6	35
0	1	0	1	15	90
0	1	1	0	13	75
0	1	1	1	22	130
1	0	0	0	8	25
1	0	0	1	17	80
1	0	1	0	15	65
1	0	1	1	24	120
1	1	0	0	14	60
1	1	0	1	23	115
1	1	1	0	21	100
1	1	1	1	30	155

Table 1 shows the values and weights of four items A, B, C, and D as an example. Table 2 lists the results of all combinations of the four items, so that the maximum result is the expected answer to the 0/1 knapsack problem.

2.2.3. The longest common subsequence (LSC) problem. The longest common sub-sequence (LCS) problem is one of the most classic examples in computer science. The purpose is to find the longest sub-sequence in two sequences and make it appear in both sequences. This problem is often applied in biology to compare DNA, RNA or protein sequences. What is special is that when dynamic programming is used to solve this problem, it can be found that its space complexity (SC) and its time complexity (TC) are equal, both are:

$$SC = TC = O(mn) \text{ (m,n are lengths of two sequences)} \tag{3}$$

DOI: 10.54254/2755-2721/71/20241687

s_1	**a**	**b**	**c**	a	d	c	c
s_2	d	**a**	a	d	**b**	**c**	d
s_3	d	c	**a**	**b**	**c**	a	

Figure 2. Example of a LCS instance with = 3 sequences s1, s2, s3 and an alphabet of Σ = {a,b,c,d}[4]

The optimal solution (sequence abc) is shown in Figure 2 in bold.

3. How dynamic programming implements and concretizes the applications above?

3.1. How it works?

3.1.1. In the shortest path problem. The Bellman-Ford algorithm is the typical way to solve this problem, which works by gradually considering whether the distance from a vertex to another vertex can be shortened by passing an intermediate point, for example, if the distance between point i and point j is greater than the sum of the distances of point i to point k and point k to point j, then update the distance of i to j.

3.1.2. In the 0/1 knapsack problem. 0/1 The knapsack problem is solved by putting or not putting items in, and it is also very straightforward to solve. According to the Knapsack Problem, it utilizes a dynamic programming table to store and update the maximum value achievable for different sub-problems. The table is filled iteratively, considering each item and its weight in turn, leading to an optimal solution, by Edwin Frank, Joseph Oluwaseyi, and Godwin Olaoye [5]. It shows the way to use dynamic programming to solve the 0/1 knapsack problem, which is like a recursive to do the same thing time by time, until getting the result.

3.1.3. In the longest common sub-sequence problem (LSC). The basic idea of the LSC problem is to use a two-dimensional array dp to store the intermediate results, where dp[i][j] represents the length of the longest common sub-string ending with X[i−1] and Y[j−1] (X and Y are two different strings).

But what needs to be noted here is that the longest common sub-sequence and the longest common sub-string are not the same thing. The former's string can be composed of discontinuous characters in the sequence, while the latter must be a continuous combination of characters.

3.2. Compare the differences of results between dynamic progremming and ordinary methods

3.2.1. The shortest path problem. (1) The common solution to the shortest path problem is a brute force solution, which means writing it down step by step directly, enumerating all possible paths and calculating the distance of each path, resulting in exponential time complexity. This method may be simpler in terms of implementation process and readability, but it will result in lower efficiency.

(2) The dynamic programming method (Bellman-Ford algorithm) is used to reduce the time complexity by trying to update the shortest paths of all edges each time, thereby improving the efficiency of the entire program and making the running speed faster.

For example, as the application of GPS (Global Positioning System) mentioned by Avinish Rai which shows how Bellman-Ford algorithm works in this problem, and it is not possible to solve this with ordinary methods. Steps below [6]:

- (1) Single Source point (H, s)
- (2) for i <--- 1 to |v[H]| - 1
- (3) do for each edge (z , x) = E[G]

DOI: 10.54254/2755-2721/71/20241687

- (4) do RELAX (z, x, y)
- (5) for each edge (z, x) = E[H]
- (6) d[x] > d[z] + w(z, x)
- (7) then return false
- (8) Return true.

3.2.2. Knapsack problem. (1) The common way to solve the knapsack problem is as A Survey of the Knapsack Problem said, when discussing the simplified version of the problem, the first answer that springs to mind is to attempt every combination and list every search space option [7]. Needless to say, its time complexity is also exponential, which is very unfriendly to efficiency and time. Referring to the examples in Table 1 and Table 2, there are already so many possible subsets for just four items; then there will be more possible subsets and the efficiency will definitely be lower if more items are processed when using this common way to solve the problem.

(2) But using dynamic programming to solve this problem will be very simple. Define the state dp[i][w] represents the maximum value of the first i items that happen to be put into a backpack with capacity w, using the state transition equation:

$$\max(\mathrm{dp}[\mathrm{i}-1][\mathrm{w}], \mathrm{dp}[\mathrm{i}-1][\mathrm{w}-\mathrm{weight}[\mathrm{i}]]+\mathrm{value}[\mathrm{i}]) \quad (4)$$

3.2.3. LSC problem. (1)The simplest way to solve this problem is to enumerate all possible substrings and check if they appear in both strings. The time complexity is usually O(n^3) (assuming the string length is n), which results in very low operating efficiency and is not easy to handle sequences with relatively large data.

(2) In dynamic programming, only a state transition equation is needed to turn a large problem into a simple, fast and intuitive solution:

$$\mathrm{dp}[\mathrm{i}][\mathrm{j}] = \mathrm{dp}[\mathrm{i}-1][\mathrm{j}-1]+1 \text{ if } \mathrm{X}[\mathrm{i}-1] = \mathrm{Y}[\mathrm{j}-1] \quad (5)$$

$$\mathrm{dp}[\mathrm{i}][\mathrm{j}] = 0 \text{ if } \mathrm{X}[\mathrm{i}-1] != \mathrm{Y}[\mathrm{j}-1] \; (!= : \text{not equal to}) \quad (6)$$

4. Gaps and improvement

4.1. Current advantages and research gaps

From the above comparison, it is found that the current advantage of dynamic programming is that, compared with brute solutions, it provides great help in reducing time complexity and improving efficiency. Despite this, dynamic programming still faces many problems. For example, first, there is no guarantee that every global optimal solution can be achieved based on the local optimal solution; additionally, it may take a long time to calculate when the state transition equation is not easy to determine or the state space is large; the most important point is that it requires additional space to store intermediate states, which may not be applicable to space complexity-sensitive problems. For example, conventional dynamic programming techniques execute in O(n^m) time, where n is the longest input string. These precise techniques quickly become unfeasible when n is large and m increases [8]; it demonstrated a situation of the example of the LSC problem when using the dynamic programming to solve it, which is fatal to the program. And there is also a shortcoming of dynamic programming regarding model development, as mentioned in Dynamic Programming, developing the right model to reflect a given circumstance is where DP gets tricky. Similar to how managing complicated problems and establishing recurrent links between linked sub-problems in successive stages requires experience in model development [9]; it proposed model development is also essential for dynamic programming, which indirectly proves that dynamic programming is not only deficient in program solutions.

DOI: 10.54254/2755-2721/71/20241687

4.2. Future development

Judging from the current development trend, we can find that dynamic programming is becoming more and more popular and practical; however, due to its shortcomings and limitations, many applications cannot be solved by it.

Based on the various algorithms that currently exist, the one that can complement dynamic programming is the greedy algorithm. The greedy algorithm can achieve the local optimal solution every time, but it does not mean that it can achieve the global optimal solution every time, and its space complexity is low, O(n); it will definitely be the most efficient, simplest and most powerful algorithm if they can combine their advantages together and get rid of their dross. Still, this risky idea won't be so ideal and successful because their weak points might appear as well. Hopefully, there will be great progress in computer science if it is successfully invented.

AI researchers interested in reinforcement learning (RL) were motivated to create stochastic versions of the value iteration and policy iteration algorithms for solving DP problems that can converge asymptotically to optimal decision rules and value functions, even for systems that are being controlled by these algorithms in real time and without the requirement for an explicit model of the structure of the Decision maker's (DM's) decision problem mentioned in Has Dynamic Programming Improved Decision Making? by John Rust [10]. In an era when AI technology is getting stronger and stronger, the future prospects of dynamic programming are still promising and important that can promote the development of AI technology.

5. Conclusion

Based on the above feedback on the comparison of different solutions and methods applied in different fields of dynamic programming, it can be seen that dynamic programming is an algorithm that can be applied to solve problems in multiple fields and plays an extremely critical role in reducing time complexity and improving algorithm efficiency. In the mean time, this article focuses on introducing and discussing three representative problems of dynamic programming in order to better understand and illustrate the operation process of dynamic programming and highlight the advantages of dynamic programming over the traditional solutions. Even though dynamic programming has a lot of excellent advantages, its bottlenecks and research gaps are obvious to be found as well when solving some problems. It is hoped that dynamic programming can be further studied in terms of space complexity in the future, so as to successfully achieve the characteristics of small space complexity, then help it become more popular, more powerful and less limited; in this way, it can have small time complexity without exchanging time for space to achieve the dual advantages of small time and space complexity, then make dynamic programming more suitable for solving more diverse problems and get more and more efficient and adaptive so that it can be used to solve some problems that it can't solve so far. As a matter of fact, solving these problems is not as simple as it sounds, but it is only a matter of time before it is achieved.

References

[1] Xiangyun Ding, Yan Gu, and Yihan Sun. 2024. Parallel and (Nearly) Work-Efficient Dynamic Programming. In Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '24). Association for Computing Machinery, New York, NY, USA, 219–232.

[2] Samah W.G. AbuSalim et al. (2020) Comparative Analysis between Dijkstra and Bellman-Ford Algorithms in Shortest Path Optimization. IOP Conference Series: Materials Science and Engineering, 917(2020) 012077.

[3] Ghadi Y.Y., AlShloul T., Nezami ZI., Ali H., Asif M., Aljuaid H., Ahmad S. (2023). An efficient optimizer for the 0/1 knapsack problem using group counseling. PeerJ Comput. Sci. 9:e1315

[4] Luc Libralesso, Aurélien Secardin, Vincent Jost. Longest common sub-sequence: an algorithmic component analysis. Hal open science, 2020. hal-02895115.

DOI: 10.54254/2755-2721/71/20241687

[5] Frank E., Oluwaseyi J., Olaoye G. (2024). Knapsack Problem. April 09. https://www.researchgate.net/publication/379652955_Knapsack_Problem
[6] Rai A. (2022) A Study on Bellman Ford Algorithm for Shortest Path Detection in Global Positioning System. International Journal for Research in Applied Science & Engineering Technology (IJRASET), Volume 10 Issue V.
[7] Assi, M., & Haraty, R. A. (2018). A Survey of the Knapsack Problem. 2018 International Arab Conference on Information Technology (ACIT), 1–6.
[8] Djukanovic M, Raidl G.R, and Blum C. "Finding Longest Common Sub-sequences: New anytime A* search results." Applied Soft Computing, Volume 95, October 2020, 106499.
[9] Ventura, J. A. (2019). Dynamic programming. Operations Research Methodologies, R. Ravindran, Ed., CRC Press, pp.168-194.
[10] Rust, J. (2019). Has Dynamic Programming Improved Decision Making? Annual Review of Economics, 11(1), 833–858.