Dynamic Programming

Thomas Neifer & Dennis Lawo

Abstract Dynamic Programming, or dynamic optimization, is an optimization approach that simplifies complex problems by breaking them into smaller, interconnected subproblems. This method eliminates redundancy and significantly improves efficiency. DP finds practical applications in various real-world problems within Operations Research, enhancing decision-making processes. Its usefulness is shown by two examples with a practical application in Python.

1 Introduction

Linear programming deals with the optimization of an objective function under certain restrictions, if these are convex (or linear). A common solution method is the simplex method, which is one of the fastest algorithms for solving such an optimization problem. Depending on the complexity, however, the runtime of the algorithm can be exponential [1]. However, it has also been shown that on average a polynomial runtime can be obtained with random input data [2].

In the context of *dynamic programming*, or *dynamic optimization*, an optimization problem is decomposed into smaller subproblems, so that the solution can be reduced from possibly exponential to merely polynomial complexity. Furthermore, dynamic optimization (DO) is not exclusively used to solve convex problems but allows the solution of various structures. For example, a linear problem can be solved by DO by decomposing it into smaller subproblems and following the corresponding DO algorithm [3]. The term dynamic programming was coined by Richard Bellman, who introduced it in the 1940s [4].

If the solution to a decision problem is assumed, in which the decisions are interdependent in time, an optimum can be found for the entire problem by means of the DO. Due to the sequential character of DO, some authors refer to a better name as stepwise or sequential optimization [5].

F. W. Peren, T. Neifer (eds.), *Operations Research and Management*,
Springer Texts in Business and Economics,
https://doi.org/10.1007/978-3-031-47206-0_11

The special property of dynamic programming is shown in its ability not to have to calculate things twice [6]. This can be illustrated by the simple example of the Fibonacci series, which is programmed exemplarily in Python. Formally, the discrete Fibonacci sequence $(0, 1, 1, 2, 3, 5, 8, 13, \dots)$ for the n-th Fibonacci number F_n can be defined as:

$$F_n = F_{n-2} + F_{n-1}$$

with start values:

$$F_0 = 0 \quad \text{and} \quad F_1 = 1$$

If we ignore the DO approach at the outset, a function for the recursive calculation of Fibonacci numbers can be formulated as follows:

Recursive Calculation of Fibonacci Numbers

```
def fibonacci(n):
    if n < 2:
       return n
    else:
       return fibonacci(n-1) + fibonacci(n-2)
```

Both initial values $F_0 = 0$ and $F_1 = 1$ are mapped here by an if condition. As long as n is less than 2, only n is returned. Otherwise, the calculation of $n \geq 2$ takes place over the repeated and additively linked call of the function, in each case for $n-1$ and $n-2$. Caused by this recursion without intermediate storage of the result, for example with the calculation of $n = 50$ through 'fibonacci(50)' again the functions 'fibonacci(49)' and 'fibonacci(49)' are executed. Furthermore, this also calls the functions 'fibonacci(48)' and 'fibonacci(47)' again for 'fibonacci(49)'. It quickly becomes apparent that a double and thus redundant call of 'fibonacci(48)' occurs here.

If this is visualized in a tree structure, the redundancy of the recursive approach without memoization becomes clear (see Figure 1).

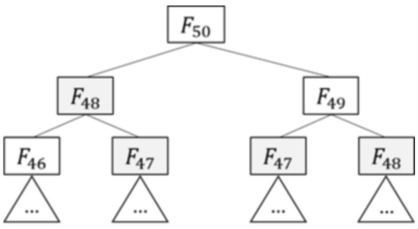

Fig. 1 Recursive Tree Structure. Source: Own representation according to Logofătu (2014) [7].

The connection of the Fibonacci series with the golden ratio (Φ) further reveals the complexity of the approach: it can be shown that the quotient of consecutive numbers of the Fibonacci sequence converges with $n \longrightarrow \infty$ to $\Phi = 1,61803..$ [8]. Accordingly, a Fibonacci number F_n, assuming that zero corresponds to the first number in the series, can be calculated as [9]:

$$F_n = \frac{\Phi^n}{\Phi + 2}$$

From this, it can be seen that the n-th Fibonacci number can essentially be described by Φ^n and the running time of the algorithm can accordingly be described by an exponential relationship [8].

Here, DO enables an enormous runtime reduction due to the polynomial relationship. This happens by the simple buffering of the determined Fibonacci value and the with a new calculation preceding the examination of whether the value is not already known. A function written in Python, which corresponds to the DO, could be designed as follows:

DO Approach with Memoization

```
dict = { }
def fibonacci(n):
    if n < 2:
       return n
    elif n not in dict:
       dict[n] = fibonacci(n-1) + fibonacci(n-2)
    return dict[n]
```

This simple memoization allows the runtime to be reduced from several years to a few milliseconds, even for $n > 100$ [7]. For the sake of completeness, the second possible solution to the problem using dynamic programming via a bottom-up approach will also be illustrated below:

Alternative DO Bottom-Up Approach

```
dict = { }
def fibonacci(n):
    dict = {0: 0, 1:1}
    for i in range(2, n + 1):
       dict[i] = dict[i-2] + dict[i-1]
    return dict[i]
```

The bottom-up approach calculates the solution to the problem from the bottom up, contrary to the recursive approach. In addition to the predefined start values for zero and one, the function is no longer called repeatedly, but the auxiliary object 'dict' is filled from bottom to top. In Operations Research (OR) there are several problems for which DO can be used. Among them are the Knapsack problem, the Traveling Salesman problem, and the order quantity problem. After the theory of DO is explained in the following, it is presented in the context of more complex and application-oriented problems of OR.

2 Theoretical Foundations

As we have already noted, DO is based on the simple premise that an optimization problem can be solved through an iterative decision process by dividing it into several subproblems. The smaller subproblems are now solved first in order to compose larger partial solutions. Only those subproblems are calculated that are actually needed to solve the larger problems. Values that have already been calculated do not have to be calculated again [10]. In this case, such a decision process is distributed over several stages (or periods), where each stage consists of a set of possible states and decisions [5].

Before the specific properties of DO models are discussed, the general form of dynamic optimization problems is described.

2.1 Definition and Properties of DO Models

The DO represents a complex procedure of the OR, which results in particular from the difficulty of the suitable modeling of the optimization problem, the design of the solution procedure as well as from stochastic influences of the problem. Therefore, there are requirements for the mathematical representation of the optimization problem, which is characterized in particular by the type of variables as well as the existing restrictions. For simplification, we will assume a minimization problem in the following, which uses an additive linkage of the stepwise objective functions. However, this can easily be transferred to a multiplicative linkage as well as to a maximization problem [5].

If the model is deterministic and discrete and is to be minimized by the sum of the stage-related objective functions, the mathematical formulation is as follows [11]:

$$F(x_1, X_2, ..., x_n) = \sum_{k=1}^{n} f_k(z_{k-1}, x_k) \rightarrow \text{min!}$$

The following restrictions must be observed [5]:

$$z_k = t_k(z_{k-1}, x_k) \quad \text{for} \quad k = 1, ..., n$$

$$z_k \in Z_k \quad \text{for} \quad k = 1, ..., n$$

$$z_0 = \alpha$$

$$x_k \in X_k(z_{k-1}) \quad \text{for} \quad k = 1, ..., n$$

where: n	:	Number of stages of a decision process
z_k	:	State of the system at the end of the stage k
Z_k	:	Set of all possible states at the end of stage k
$z_0 = a$	:	Initial state
Z_n	:	Set of possible or given final states after n stages
x_k	:	Decision in stage k
$X_k(z_{k-1})$	:	Set of all decisions that can be selected in stage k starting from state $k - 1$
$t_k(z_{k-1}, x_k)$	:	Transformation function that defines the state z_k of which the system transitions in stage k after the decision x_k is made at the end of stage $k - 1$ in state z_{k-1}.
$f_k(z_{k-1}, x_k)$	:	Stage-related objective function, which describes the influence of the decision x_k made in dependence on the state z_{k-1} on the objective function value. In general, for each DO problem it must hold that f_k depends only on the state z_{k-1} of the preliminary stage and the decision x_k. This corresponds to the Markov property [11], [12].

DO models can be classified according to the following criteria (see Table 1 [5]:

Table 1 DO Model Properties

Time intervals of the periods or stages	Distinction between discrete and continuous models. While discrete models represent changes of state at the point in time or in discrete steps, continuous models allow continuous changes of state.
Disturbance variable	In deterministic models, the disturbance variable b_k (external influences on the model) can only assume exactly one value. In stochastic models, however, b_k itself represents a random variable, which means that it can assume different values with known probabilities.
State and decision variables	State and decision variables can be single-valued or multi-valued in the form of vectors.
Finiteness of state space	Z_k and X_k can be either finitely bounded or infinite with respect to their set.

Source: Own representation according to Domschke et al. (2015) [5].

Figure 2 illustrates the causalities between the above terms once again.

Discrete and deterministic DO models can be clearly represented in a digraph $G = (V, E)$ (see Figure 3) [5]. A digraph, or directed graph, is composed of a finite, nonempty set of nodes V and a set of arrows E. The elements of E are also called directed edges [14]. Here, V represents the union set of all Z_k for $k = 1, \ldots, n$, where

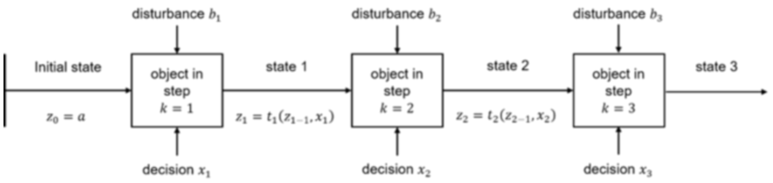

Fig. 2 Graphical Representation of a DO Model. Source: Own representation according to Schwarz (2010) [13].

each node $z_k \in Z_k$ describes an acceptable state. E consists of arrows (z_{k-1}, Z_k) with $z_{k-1} \in Z_{k-1}$ as well as $z_k \in Z_k$ for $k = 1, \ldots, n$. An arrow represents the system transition from a state z_{k-1} to z_k, which can be explained by $z_k = t_k(z_{k-1}, x_k)$. Due to the deterministic model assumption, the explicit specification of b_k is omitted here, since it is a constant.

With each transition from z_{k-1} to z_k, the objective function is influenced, because with each state change period-related costs are associated, the amount of which is determined by the stage-related objective function $f_k(z_{k-1}, x_k)$.

Furthermore, $z_0 = \alpha$ as the initial state and $z_n = 0$ as the final state applies as usual [5].

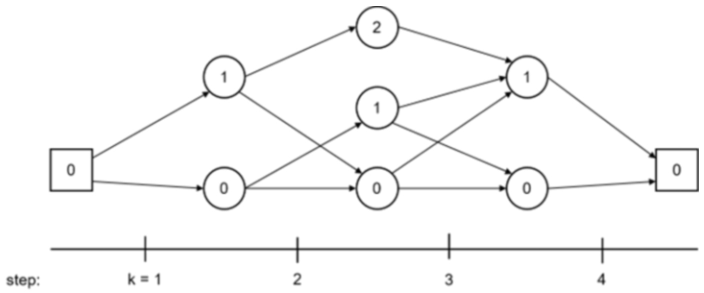

Fig. 3 Digraph of an Exemplary DO Model. Source: Own representation according to Domschke et al. (2015) [5].

2.2 Solution Principle of Dynamic Optimization

Finding an optimal policy for a discrete and deterministic DO model with initial state $z_0 = \alpha$ and a set Z_n of possible final states is called problem $P_0(z_0 = \alpha)$. Similarly, the task of determining an optimal policy for transforming a state $z_{k-1} \in Z_k$ into one of the possible final states $\in Z_n$ is defined as problem $P_{k-1}(z_{k-1})$. Here, the optimal objective function value of a problem $P_k(z_k)$ is $F_k^*(z_k)$.

Here, a policy denotes a sequence of decisions $(x_j, x_{j+1}, ..., x_k)$ that transforms a system to a state $z_k \in Z_k$ given a state $z_{j-1} \in Z_{j-1}$. Thus, assuming a minimization problem, a sequence of decisions $(x_j^*, x_{j+1}^*, ..., x_k^*)$ is called an optimal policy if it transforms a system from a given state $z_{j-1} \in Z_{j-1}$ to a state $z_k \in Z_k$ while minimizing an objective function [11].

In order to determine an optimal overall policy of a system, the following applies in accordance with the optimality principle going back to Bellman (1957) [15]:

Theorem 0.1 *Let* $(x_1^*, ..., x_{k-1}^*, x_k^*, x_n^*)$ *be an optimal policy that transforms a system from an initial state* $z_0 = \alpha$ *to a final state* z_n *and let* z_{k-1}^* *be a state that the system reaches at stage* $k-1$ *then follows:*

$(x_k^*, ..., x_n^*)$ *is an optimal (partial) policy which, starting from the state* z_{k-1}^* *in stage* $k-1$*, transforms the system to the given or allowed final state* z_n *(backward recursion) and* $(x_1^*, ..., x_{k-1}^*)$ *is an optimal (partial) policy that transforms the system from a given initial state* $z_0 = \alpha$ *to a state* z_{k-1}^* *in stage* $k-1$ *(forward recursion) [5].*

Accordingly, an optimal policy exists exactly when each sub-policy is also optimal. The DO uses Bellman's optimality principle to derive an optimal overall policy by forward or backward recursion [3].

2.3 Bellman's Functional Equation Method

Bellman's Functional Equation Method describes a methodology for process analysis and optimization, which consists of the phases of decomposition, backward, and forward recursion. In decomposition, the decision process is decomposed into several sub-problems, where only the decision options are considered. By alternating the solution of the sub-problems or their alternative decomposition into further sub-problems, the smallest sub-problems are identified and finally solved. Thus, in the next step, the solutions for the next larger (sub-)problems are created. By means of backward recursion, optimal decisions of all intermediate states of the (sub)problems are now made backward, taking into account the objective function, with the final state as the starting point. Forward recursion, on the other hand, considers the existing initial state as the starting point. The optimal decisions made under the objective function are made on the basis of those decisions made in the previous backward recursion [16].

The goal is to observe a trajectory $(x_0, x_1, ..., x_n)$ which satisfies the optimality principle. For this Bellman defines the following equation:

$$S(x) = \min_{y \in U(x)} g(x, y) + S(y)$$

where the value of the trajectory is defined as the summed value of the different sub-problems. This means that:

$$g(x_i, x_{i+1}, ..., x_j) = \sum_{k=i}^{j-1} g(x_k, x_{k+1})$$

where: x	:	Finite set of possible system states
$x_i, i = 0, ..., n$	:	System States subdivided into several successive sub-problems x
$U(x)$	:	Set of subsequent states for each system state x in the frame of stages 0 to $n - 1$
$(x_i, x_{i+1}, ..., x_j)$	:	State sequences (trajectories), which allow transitions between the states
$0 \leq g(x_i, x_{i+1}, ..., x_j)$	:	Non-negative evaluation function for the trajectories and their sections
S	:	Optimal value that can be assigned to each state, where the target set x_n contains at least one optimal value
$S(x)$	:	Value of an optimal trajectory. Assumes the value ∞ if no trajectory is present [16].

The Bellman-Ford algorithm thus works simplistically according to the following principle (pseudocode) [17]:

```
Step 1: Assign $x_i$ to infinity and $x_0$ to zero for $i \neq 0$
Step 2: for each edge$(u, v)$ do $n - 1$ times:
            $x_i = \min\{x_i, x_u + \text{weight}_{uv}\}$
Step 3: for any edge(u, v):
            if $x_u + \text{weight}_{uv} < x_i$:
                negative-weight cycle
```

3 Applications

3.1 Basic Example: Finding the Shortest Route

Let us assume that there are seven cities A to G (nodes), which have the following traffic connections (edges) including the specified distances to each other (see Table 2):

If this is visualized by means of a graph, Figure 4 results.

Table 2 Distances Between the Seven Cities

From	To	Distance
A	B	17
A	C	17
A	D	1
A	E	10
A	G	2
B	D	6
B	E	16
C	F	5
C	G	7
D	F	7
F	E	18
F	G	8

Source: Own representation.

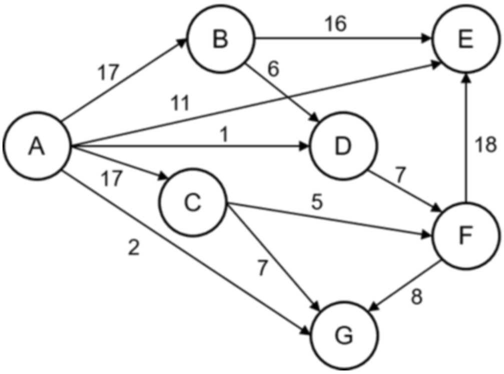

Fig. 4 Graph: Distances Between the Seven Cities. Source: Own representation.

Initiation and Phase 1:

The goal is to find the shortest distance from the starting point A to the other cities. Let us consider the example with A as the starting point. Accordingly, the worst estimate applies for $k = 0$, where A is assigned the value 0 and B to G is assigned the value infinity.

In phase 1, all edges and the respective distances to the respective city are assigned as well as the predecessor, i.e. the city crossed before. For A, the value 0 is still used, since this is the starting point and no distance has to be covered.

The first edge is now A $\rightarrow$ B, which has a distance of 0 (coming from A) + 17 (to B). Since 17 is less than infinity, the new value is assigned, as well as the information that the predecessor node is A. The next edge, B $\rightarrow$ E, includes a distance of 16. Thus, starting from the starting point A, E is 17 (to B) + 16 away, making a total of 33. Since 33 is less than infinity, the new value is assigned as well as B as the predecessor node. The edge B → D is treated analogously: Starting from A, 17 + 6 and consequently 23 are needed to get to D via B. 23 is less than infinity and is therefore assigned as the new value. Since, in addition to A → B → E, there is also a direct connection between A and E, which, at 11, is smaller than 33, the smaller value is used here, starting from A. The same applies to the following edge A → D: Since 1 is smaller than 23, the new value is used.

Edge D → F requires 1 (from A to D) + 7 (from D to F) and therefore 8. Since the edge F → E comprises a total of 26 (8 from A to F + 18 from F to E) and this is more than 11 (A → E), 11 remains.

The remaining edges are treated in the same way, resulting in Table 3.

Table 3 Finding the Shortest Route: Phase 0 and 1

k	A	B	C	D	E	F	G
0	0	∞	∞	∞	∞	∞	∞
1		17 A	17 A	~~23 B~~ 1 A	~~33 B~~ 11 A	8 D	2 A

Source: Own representation.

Phase 2:

Phase 2 now iterates over the edges and nodes again and checks in an analogous way whether an improvement of the previous results occurs. Since this is not the case, the minimum distances of the cities to the starting point A were identified (see Table 4).

Table 4 Finding the Shortest Route: Phase 2

k	A	B	C	D	E	F	G
0	0	∞	∞	∞	∞	∞	∞
1	0	17 A	17 A	1 A	11 A	8 D	2 A
2	0	17 A	17 A	1 A	11 A	8 D	2 A

Source: Own representation.

3.2 Bellman-Ford Algorithm in Python

Next, let's look at how an algorithm can solve the example from chapter 3.1. For this purpose, we define the problem from Figure 4 again as input for the algorithm:

Initiation of Nodes and Edges

```
# Definition of nodes A - G as a list (A = 0, B = 1, etc.):
nodes = [0, 1, 2, 3, 4, 5, 6]

# Definition of edges as dictionary:
edges = {(0,1): 17, (0,2): 17, (0,3): 1, (0, 4): 11, (0, 6): 2,
               (1,3): 6, (1,4): 16,
               (2,5): 5, (2,6): 7,
               (3,5): 7,
               (5,4): 18, (5,6): 8}
```

Then, a function is defined that reflects the Bellman-Ford algorithm:

Definition of the Bellman-Ford function

```
# Definition of the Bellman-Ford function. The required arguments are the
# nodes, the edges with their respective weights as well as the start node.

def bellmanford(nodes, edges, sourceNode = 0):
    # Initiation: start node is assigned 0, the rest infinite
    pathDistances = {v: float('inf') for v in nodes}
    pathDistances[sourceNode] = 0
    # Path definition
    paths = {v: [] for v in nodes}
    paths[sourceNode] = [sourceNode]
     # Bellman-Ford algorithm:
    for _ in range(len(nodes) - 1):
        for (u, v), w_uv in edges.items():
            if pathDistances[u] + w_uv < pathDistances[v]:
                pathDistances[v] = pathDistances[u] + w_uv
                paths[v] = paths[u] + [v]
    return pathLengths, paths
```

The determination and output of the result are as follows:

Apply the Bellman-Ford function

```
shortestDistances, shortestPaths = bellmanford(nodes, edges)

print(shortestDistances)
print(shortestPaths)
```

The output is now analogous to the above result from chapter 3.1:

```
print(shortestDistances)
{0: 0, 1: 17, 2: 17, 3: 1, 4: 11, 5: 8, 6: 2}
print(shortestPaths)
{0: [0], 1: [0, 1], 2: [0, 2], 3: [0, 3], 4: [0, 4], 5: [0, 3, 5], 6: [0, 6]}
```

For example, to get from the starting point 0 (A) to point 5 (F), a distance of 8 (0 + 1 + 7) must be covered. This corresponds to the path [0, 3, 5] and therefore A → D → F.

4 Conclusion

Dynamic programming is an important tool for breaking down complex real-world optimization problems into small, manageable sub-problems. The Bellman-Ford algorithm makes it possible to achieve optimization by an iterative approach. Recursion is of essential importance, but it becomes more and more time and resource-consuming. For this purpose, the memorization of the subproblem solutions serves to counteract this weakness. All in all, dynamic optimization is suitable for the optimization of many different real-world problems and thus represents a powerful tool for operation research.

References

[1] V. Klee and G. J. Minty, “How good is the simplex algorithm,” Inequalities, vol. 3, no. 3, pp. 159–175, 1972.
[2] K.-H. Borgwardt, “The average number of pivot steps required by the simplex-method is polynomial,” Zeitschrift für Operations Research, vol. 26, no. 1, pp. 157–177, 1982.
[3] H. Baumann et al., “Dynamische Optimierung,” in Lehrbuch der Mathematik für Wirtschaftswissenschaften, H. Körth, C. Otto, W. Runge, M. Schoch, and W. Adler, Eds. Wiesbaden: VS Verlag für Sozialwissenschaften, 1975, pp. 697–723. doi: 10.1007/978-3-322-87545-7_13.
[4] R. Bellman, “Dynamic programming,” Science, vol. 153, no. 3731, pp. 34–37, 1966.
[5] W. Domschke, A. Drexl, R. Klein, and A. Scholl, Einführung in operations research. Springer-Verlag, 2015.
[6] F. Gurski, I. Rothe, J. Rothe, and E. Wanke, Exakte algorithmen für schwere graphenprobleme. Springer-Verlag, 2010.
[7] D. Logofătu, Grundlegende Algorithmen mit Java: Lern- und Arbeitsbuch für Informatiker und Mathematiker. Springer-Verlag, 2014.
[8] T. von Brasch, J. Byström, and L. P. Lystad, “Optimal Control and the Fibonacci Sequence,” Journal of Optimization Theory and Applications, vol. 154, no. 3, pp. 857–878, Sep. 2012, doi: 10.1007/s10957-012-0061-2.
[9] G. B. Meisner, The Golden Ratio: The Divine Beauty of Mathematics. Race Point Publishing, 2018.

[10] D. Logofătu, “Dynamische Programmierung,” in Grundlegende Algorithmen mit Java, Wiesbaden: Vieweg, 2008, pp. 231–310. doi: 10.1007/978-3-8348-9433-5_8.
[11] S. Dempe and H. Schreier, Operations research: deterministische modelle und methoden. Springer-Verlag, 2007.
[12] M. Frydenberg, “The chain graph Markov property,” Scandinavian Journal of Statistics, pp. 333–353, 1990.
[13] C. Schwarz, Effiziente Algorithmen zur optimalen Lösung von dynamischen Losgrößenproblemen mit integrierter Wiederaufarbeitung. Berlin: Logos-Verl, 2010.
[14] W. H. Janko, Informationswirtschaft 1: Grundlagen der Informatik für die Informationswirtschaft. 1998. Accessed: Jun. 22, 2020. [Online]. Available: http://link.springer.com/openurl?genre=bookisbn=978-3-540-64812-3
[15] B. Richard, “Dynamic programming,” Princeton University Press, vol. 89, p. 92, 1957.
[16] M. Papageorgiou, M. Leibold, and M. Buss, “Dynamische Programmierung,” in Optimierung, Springer, 2015, pp. 357–386.
[17] D. Walden, “The bellman-ford algorithm and ‘distributed bellman-ford’” 2005.