Dynamic financial withdrawal planning from interest-bearing assets

Dynamické plánovanie výberu financií z úročených aktív

Daniel Dudek[1]

Abstract
Dynamic programming offers tools to mathematically describe the effects of continuously executed processes. Policymakers or Process owners in companies could use dynamic programming to find the optimal use of such processes. This article aims to show the use of dynamic programming to develop a plan for the dynamic withdrawal of funds from interest-bearing assets.

Key words
Dynamic Programming, Plan Creation, Reproduction Model, Policy Making

Abstrakt
Dynamické programovanie ponúka nástroje na matematický opis účinkov nepretržite vykonávaných procesov. Tvorcovia politík alebo vlastníci procesov v podnikoch by mohli dynamické programovanie využívať na hľadanie optimálneho využitia takýchto procesov. Cieľom tohto článku je ukázať využitie dynamického programovania za účel vytvoreniu plánu pre dynamický výber financií z úročiacich aktív.

Kľúčové slová
dynamické programovanie, tvorba plánu, reprodukčný model, tvorba politík

JEL classification
C61

1 Introduction

In this article, we will show the utilisation of Bellman Dynamic Programming on simple problems. The basic idea of Dynamic Programming is to solve a problem using a divide-and-conquer approach wherein the solutions of overlapping subproblems are reused to avoid recalculating solutions. Toth Horowitz and Sahni presented an improved dynamic programming algorithm for solving the knapsack problem (Chebil & Khemakhem, 2015). Generally, dynamic programming-based algorithms are efficient and easy to implement, particularly for small and medium-sized instances. We will show some applications of such matter in various problems across the world. In the end, we developed mathematical model inspired by reproduction model that create plan that will maximise the profit with highest liquidity form.

2 Literature review

The main idea of Dynamic Programming (DP) is to decompose the problem into more manageable subproblems. Computations are then carried out recursively, where the optimum solution of one subproblem is used as an input to the next subproblem. The optimum solution for the entire problem is at hand when the last subproblem is solved. How the recursive computations are carried out depends on how the original problem is decomposed. In particular,

[1] Bratislava University of Economics and Business, Faculty of Economic Informatics, Department of Operations Research and Econometrics, Dolnozemská cesta 1, 852 35 Bratislava, daniel.dudek@euba.sk.

the subproblems are usually linked by typical constraints. The feasibility of these common constraints is maintained at all iterations (Taha et al., 2017).

The central recursive equation expresses the shortest distance $f_i(x_i)$ at stage i as a function of the next node x_{i+1}. Here, i ranges over the finite set of stage indices $I=\{0,1,\ldots,N\}$, so each value of i labels a specific decision stage in the process. In dynamic programming terminology x_i is the state at stage i. The state links successive stages in a way that allows optimal decisions at a future stage to be made independently of all choices at preceding stages. Defining the state in this manner leads to the following unifying framework for dynamic programming (Sieniutycz & Jeżowski, 2013).

Future decisions for all future stages constitute an optimal policy regardless of the policy adopted in all preceding stages. The principle of optimality does not address how a subproblem is optimised. The reason is the generic nature of the subproblem. It can be linear or nonlinear, and the number of alternatives can be finite or infinite. All the principle of optimality does is "break down" the original problem into more computationally tractable subproblems (Taha et al., 2017).Table 1 crystallises the standard finite-horizon dynamic-programming framework by cataloguing the temporal index, state and control manifolds, admissible boundary sets, cost functionals, and deterministic state-transition operator that underpin the subsequent analytical developments.

Tab. 1: Description of the elements of dynamic programming definition

Symbol	Meaning
T	The finite planning horizon, i.e., the number of stages (time-steps) in the decision process. Stages are indexed $t=0,1,\ldots,T$. In a finite-horizon problem T is fixed; in an open (indefinite) horizon it may vary or be chosen optimally.
t	Continuous time variable or in other words stage, $t\in[0,T]$.
Q	The action (control) set – all decisions $q(t)$that are admissible at any stage t**.**
X	The state space – all system states $x(t)$that are allowed during the horizon**.**
$P\subseteq X$	The set of admissible initial states. At $t=0$ the process must start in one of these states.
$C\subseteq X$	The set of admissible terminal (goal) states. At the final stage $t=T$ the state must lie in this set.
$x(t)$	The state variable at stage t, taking values in X**.**
$q(t)$	The control (decision) variable at stage t, taking values in Q**.**
$f(x,q)$	The stage-cost (or reward) function incurred when the state is x and the action q is applied.
$f(t,x,q)$	Instantaneous cost (or reward) density incurred at time t when the state is x and the control is q.
$g(t,x,q)$	State-transition (dynamics) function giving (time derivative) $\dot{x}(t)=g(t,x(t),q(t))$

Source: Processed from (Laščiak et al., 1983)

There are two types of optimisation methods based on dynamic programming that are Discrete dynamic optimisation and continuous dynamic optimisation.

- Discrete dynamic optimisation, which tries to find such policy from all the possible policies (Laščiak et al., 1983).

The goal is to find a sequence of controls $\{q(t)\}_{t=0}^{T-1}$ such that

Controls are noted as:

$$q(t) \in Q, \qquad \text{for } t = 0,1,\dots,T-1 \tag{1}$$

States are noted as:

$$x(t) \in X, \qquad \text{for } t = 0,1,\dots,T \tag{2}$$

Initial state is noted as:

$$x(0) \in P \tag{3}$$

Terminal state is noted as:

$$x(T) \in C \tag{4}$$

and the cumulative cost:

$$\sum_{t=0}^{T-1} f\big(x(t), q(t)\big) \tag{5}$$

is minimised (or maximised, depending on the formulation).

- Continuous dynamic optimisation problem, which tries to find such policy from all the possible policies, that finds a control trajectory $\{q(t)\}_{t\in[0,T]}$ where (Laščiak et al., 1983):

Controls are noted as:

$$q(t) \in Q, \quad t \in [0,T] \tag{6}$$

States are noted as:

$$x(t) \in X, \qquad t \in [0,T] \tag{7}$$

Initial state is noted as:

$$x(0) \in P \tag{8}$$

Terminal state is noted as:

$$x(T) \in C \tag{9}$$

subject to the system dynamics

$$\dot{x}(t) = g(t, x(t), q(t)), \qquad t \in [0,T] \tag{10}$$

and cumulative COST, that represents cost regarding the optimisation

$$COST = \int_0^T f\big(t, x(t), q(t)\big)dt \tag{11}$$

is minimised (or maximised, depending on the formulation).

There exists two horizon types (Carlson et. al., 1991):

- Finite-time horizon: T is prescribed and constant; optimisation is performed over the fixed interval $[0,T]$.
- Open (indefinite) horizon**:** T is not predetermined; it may depend on the policy or be itself a decision variable, leading to problems where the optimal stopping time is part of the solution.

Another characteristic of the dynamic programming approach is developing a recursive optimisation procedure, which builds to a solution of the overall N-stage problem by first solving a one-stage problem, sequentially including one stage at a time, and solving one-stage problems until the overall optimum has been found. This procedure can be based on a backward induction process, where the first stage to be analysed is the final stage of the problem, and problems are solved, moving back one stage at a time until all stages are included. Alternatively, the recursive procedure can be based on a forward induction process, where the first stage to be solved is the initial stage of the problem, and problems are solved moving forward one stage at a time until all stages are included. In specific problem settings, only one of these induction processes can be applied (e.g., only backward induction is allowed in most problems involving uncertainties). The basis of the recursive optimisation procedure is the so-called principle of optimality, which has already been stated: an optimal policy has the property that, whatever the current state and decision, the remaining decisions must constitute an optimal policy about the state resulting from the current decision (Massachusetts Institute of Technology, 2015). Those methods are:

1. Top-down method
2. Bottom-up method

The top-down method solves the overall problem before breaking it into subproblems. This process solves more significant problems by recursively finding the solution to subproblems, caching each result. This memorisation process helps avoid solving the problem repeatedly if it is called more than once. The top-down method, can return the result saved as it was solved in the context of the overall problem, thus storing the results of already solved problems. The most common related process is called forward induction (Jaffar et al., 2008).

The bottom-up—or tabulation—method works in the opposite direction. It evaluates **all** sub-problems in an order that guarantees every dependency is already known when needed, then stores those answers in a table. Because there may be many indices (stage, capacity, remaining time, etc), the table is typically n-dimensional, where n represents the number of independent indices that characterise a sub-problem, while ($n \geq 1$). Earlier we used the capital N for "the number of stages" in a discrete model. Here the lower-case n simply counts how many indices are required to label sub-problems; it is unrelated to N. Once every entry of the table is filled, the value of the original problem is read directly from the appropriate cell. Computing the table from the "end" of the decision process toward the "start" is known as backward induction (Wimmer et al., 2018).

Forward and backward induction always return the same optimal value. Nevertheless, most dynamic-programming textbooks and software libraries default to backward induction because, in many practical models, it stores fewer intermediate states and therefore runs faster (Taha et al., 2017).

Backward induction is determining a sequence of optimal choices of action by employing reasoning backwards in sequence, from the end of a problem or situation to its beginning, choice by choice. It proceeds by examining the last point at which a decision is to be made and then identifying the most optimal choice of action. Using this information, one can determine what to do at the second-to-last point of the decision. This process continues backwards until one has determined the best action for every possible point along the sequence (Matias et al., 2023).

However, the Bellman optimality Principle is the method that made dynamic programming a respected part of mathematics. Bellman's optimality principles are suitable for optimal conditions for inherently discrete processes. Nevertheless, under the differentiability assumption, the method only enables an easy passage to its limiting form for continuous systems. The application of the method is straightforward when it is applied in the optimisation of control systems without feedback. Dynamic programming (DP) is crucial for the optimal performance potentials discussed in this book and for deriving pertinent equations that describe these potentials. The DP method is based on Bellman's principle of optimality. It makes it possible to replace the simultaneous evaluation of all optimal controls with sequences of local evaluations at sequentially included stages for evolving subprocesses (Matias et al., 2023).

Description of the elements in Tab. 2 and in formulas below are completely separated from previous notations of elements and formulas.

Tab. 2: Description of the elements of Bellman's Expectation Equation in deterministic cases

Symbol	Definition
s	A state the agent can occupy.
a	An action chosen in state s.
s'	The next state reached after taking action a.
$V(s)$	The value (expected return) associated with state s.
$R(s,a)$	The immediate reward received when action a is executed in state s.
$\gamma \in [0,1)$	The discount factor that down-weights future rewards.
$\pi=\{\pi_1,\pi_2,\ldots,\pi_I\}$	A finite collection of candidate policies (instruction sets) among which we search for the optimal one.
I	The number of policies in that set. We enumerate the options as $\pi_1,\pi_2,\ldots,\pi_I$ to keep track of each distinct strategy.

Source: Processed from (Bellman, 1954)

The most basic Bellman's Expectation Equation for optimising value can be stated as (Bellman, 1954):

$$V(s) = \max_a \big(R(s,a) + \gamma V(s')\big) \tag{12}$$

During backward (bottom-up) induction the agent starts at the goal, assigns terminal values, and then works backward through the state space, filling in $V(s)$ by applying the Bellman equation. At each step it picks the action a that maximises the bracketed expression, thereby choosing the policy π_i that ultimately delivers the highest cumulative reward. In Tab.3, is depicted the exemplar solution where $\gamma = 0.9$:

Tab. 3: Example of space sweeping by agent

V=0.81 (0+0.9*(0.9)) $a^* = \rightarrow$	V=0.9 (0+0.9*(1)) $a^* = \rightarrow$	V=1 (1+0.9*(0)) $a^* = \rightarrow$	R=+1 (Goal)
V=0.73 (0+0.9*(0.81)) $a^* = \uparrow$	R=-M	V=0.9 (0+0.9*(1)) $a^* = \uparrow$	R=-1
V=0.66 (0+0.9*(0.73)) $a^* = \uparrow$ or $\rightarrow$ (Start)	V=0.73 (0+0.9*(0.81)) $a^* = \rightarrow$	V=0.81 (0+0.9*(0.9)) $a^* = \uparrow$	V=0.73 (0+0.9*(0.81)) $a^* = \leftarrow$

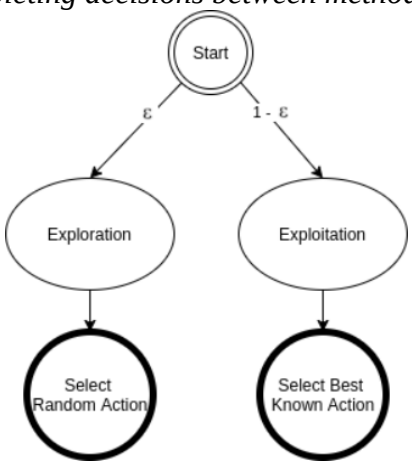
Source: Own Elaboration

When the agent calculates all the known $V(s)$ values, it can use a greedy algorithm to find the policy, leading him from the start to the goal. After the application of the greedy algorithm, the resulting optimal policy will be: $\pi_i = \uparrow\uparrow\rightarrow\rightarrow\rightarrow$ or $\rightarrow\rightarrow\uparrow\uparrow\rightarrow$

In dynamic programming, while we use the bellman method with the top-down method, we select a policy based on its reward. The agent always chooses the optimal action. Hence, it generates the maximum reward possible for the given state. In our problem, we will use this greedy algorithm, named the economic strategy (Baeldung, 2023).

If we used the tree diagram to calculate all the possibilities of the problem for the given number of iterations, we could use backward induction to find the optimal solution where we are exercising the reward (or payoffs) without the need for exploration. This effect is obtained because backward induction has the final values of all previous policies, so it can more easily determine the optimal action from all possibilities.

Fig. 1: Diagram depicting decisions between methods in DP of the agent



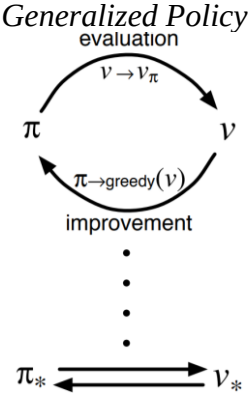
Source:(Baeldung, 2023)

In epsilon-greedy action selection, the agent uses both pathways, exploitation, to take advantage of prior knowledge and exploration to look for new options. Converging policy evaluation could achieve a similar effect.

We can use both Bellman equations to find a solution that will converge to the optimal solution by the policy and the value. This converging solution principle is called Generalized Policy Iteration (GPI), defined as any interaction of policy evaluation and policy improvement, independent of their granularity. In Fig 2, the GPI works as depicted in the conceptual model, that means the GPI is constantly switching between finding the optimal policy π^* (a decision

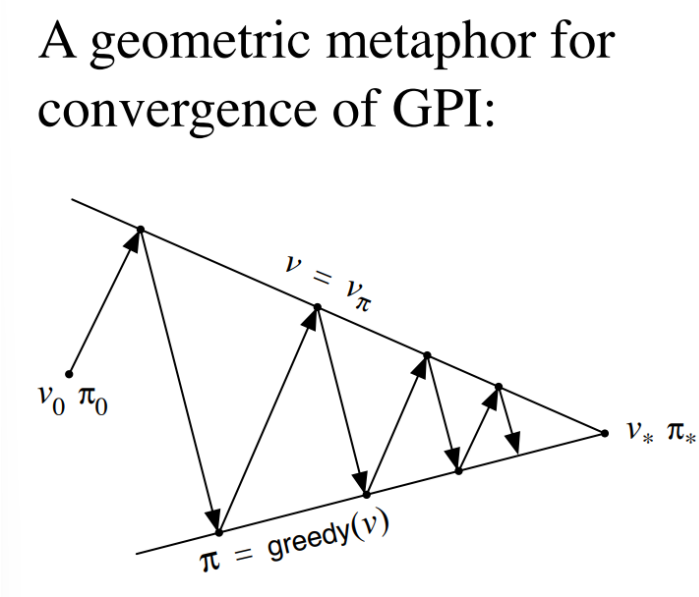
rule that maximises expected return from every state,) and the optimal state-value function v^*. Fig 3 depicts those two improving evaluations' convergence to the optimal solution.

Fig. 2: Depicment of Generalized Policy Iteration Mechanism



Source: (Sutton et al., 2018)

Fig. 3: Convergence to most effective solution possible



Source: (Sutton et. al., 2018)

Dynamic programming is an excellent tool with wide application in various management optimisation problems. The first reason is that dynamic programming is centred around the effects of time. As we said before, ecological activities require some time to take effect, while this effect can be halted or boosted through additional activities in the time window. The second parameter, which could modify Bellman's optimisation equations, is stochastic values. In environment management, it is impossible to perfectly predict all outcomes and all the effects that could show up through the optimising process.

There are many models regarding dynamic programming. The most famous methods are:

- Workforce size model, where there could be a construction project that runs week by week. For every week the site manager knows the minimum number of workers that must be on the job to meet schedule targets. The company can raise the crew above that minimum by hiring extra people and can reduce it by letting workers go (So & Kek, 2020).

  - Holding extra workers**.** Keeping more workers than the minimum costs money (overtime, idle time, salaries, benefits, and so on). The larger the surplus, the higher the weekly "holding" expense.
  - Hiring new workers. Bringing additional people onto the crew from one week to the next triggers a separate "hiring" cost that covers recruiting, onboarding, and training.
  - Firing workers. In this simplified version of the model, letting employees go is assumed to be free; no severance or rehiring penalty is counted.

  The planner's task is to decide, for each week of the project, how large the crew should be so that the total cost—holding extra staff plus any hiring charges—is as small as possible while never dropping below the required minimum. Dynamic-programming techniques can then be applied to find the cost-minimising schedule (Taha et al., 2017).
- Equipment- replacement model, that addresses that machines become more expensive to own the longer they remain in service: breakdowns grow more frequent, maintenance bills rise, and they earn less income. At some point it is cheaper to scrap an aging unit and buy a new one. The equipment-replacement model helps a manager decide exactly when that should happen over a planning horizon of several years. The life cycle of machines could be described in this model as (Lu & Wang, 2013) :
  - Operating year by year. At the start of every year, decision-maker must choose one of two actions:
    - Keep the current machine for another year, accepting its expected revenue and its operating and maintenance cost for that age
    - Replace it immediately with a brand-new unit, paying the purchase price and then earning the revenue and paying the costs associated with a first-year machine.
  - Age-dependent figures. For any given age of the machine the model tabulates three numbers:
    - the income the machine is expected to generate during that year,
    - the expense of running and maintaining it,
    - the amount that could be recover by selling or scrapping it at that moment (its salvage value).
  - Purchase cost**.** Buying a new machine always requires the same upfront investment, regardless of the calendar year in which it was done.

  By comparing the stream of cash flows that results from "keep" versus "replace" decisions in every possible year, the model reveals the most economical schedule: how long to hold on to each machine before swapping it for a new one so that total profit over the entire planning period is maximised (or total cost is minimised). This approach follows the dynamic-programming treatment described by Taha et al. (2017).
- Investment-allocation model, that models situation where an investor plans to inject predetermined sums of cash at the beginning of each of the next several years. For every new deposit the investor can choose between banks, that offer different outcomes. To attract business, each bank also pays a bonus on fresh deposits. The bonus is calculated as a fixed percentage of that year's new money, and the percentage can vary from year to year and from one bank to the other. Possible key rules of the scheme could be (Yu & Kuang, 2015):
  - Timing of bonuses**.** The bonus for a given deposit is credited at the **end** of the same year in which the deposit was made.

- Reinvestment options**.** At the start of the following year the investor may allocate that bonus—together with the next year's scheduled cash contribution—to either bank, again earning interest and (if it counts as "new" money) another bonus.
- Lock-in period for principal**.** Once a deposit is placed in a bank, the principal must stay there until the final year of the multi-year horizon; it cannot be moved between banks or withdrawn early.

The decision problem is to work out, year by year, which bank should receive each scheduled deposit and each newly earned bonus so that the total wealth at the end of the planning horizon is maximised. This dynamic-programming formulation is adapted from the treatment in Taha et al. (2017).

- Inventory Models*:* DP has essential applications in inventory control.
- Reproduction Model
- Several algorithms emerged from the definition of dynamic programming, for example:
- Djikstra algorithm: for the shortest path. Dijkstra's algorithm, published in 1959 and named after its creator, Dutch computer scientist Edsger W. Dijkstra, can be applied to a weighted graph. The graph can be either directed or undirected. One stipulation to using the algorithm is that the graph needs to have a nonnegative weight on every edge (Abiy et al., 2017).

Dynamic programming is an excellent tool, even for more pressing problems that are now even more prominent than ever, such as environmental management. The first reason is that dynamic programming is centred around the effects of time (Munch & Brias, 2024). As we said before, ecological activities require some time to take effect, while this effect can be halted or boosted through additional activities in the time window. The second parameter, which could modify Bellman's optimisation equations, is stochastic values (Davidsen et al., 2015). In environment management, it is impossible to perfectly predict all outcomes and all the effects that could show up through the optimising process.

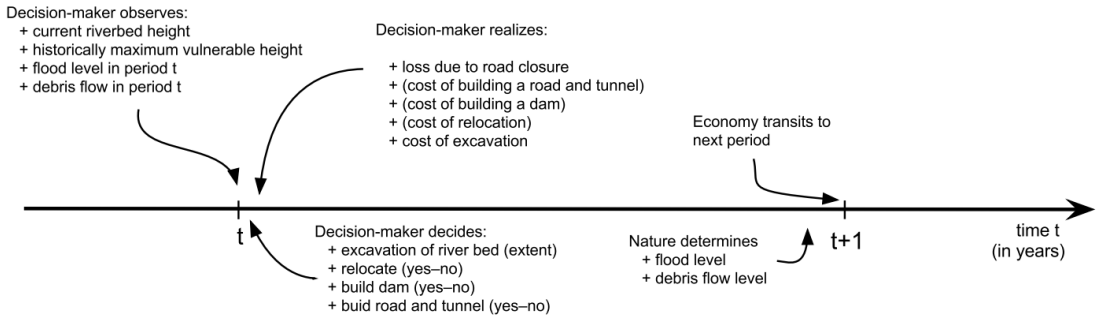
Several decision-support tools in environmental management exist to deal with different degrees of uncertainty. Cost-benefit analysis is a relatively common decision tool to inform adaptation employed for deterministic analysis, where variables such as costs and benefits of projects/programs are known and can be compared to justify interventions based on an efficient allocation of resources. At the other end of the spectrum, tools support decision-making under deep uncertainties (DMDU), where many plausible futures are possible, and a broad range of solutions or outcomes exist. DMDU tools such as robust decision-making, dynamic adaptive policy pathways and accurate options analysis can be handy in these cases (Muccione et al., 2023).

Uncertainties might not be profound but instead arise from a lack of information. In these cases, uncertainties can be dealt with by using iterative risk management approaches, which allow the integration of learning processes into decision-making cycles. Likewise, stochasticity or randomness can be appropriate representations for uncertainties in systems whose input parameters can be described probabilistically. Recent advances in computational economics have developed and applied methods of numerical dynamic programming that integrate stochastic or random uncertainties into the decision-making Natural Hazards processes at relatively low computational costs. Dynamic programming has been successfully employed in environmental decision-making to solve stochastic problems. It is a well-established approach in environmental management and the context of water resource management. Dynamic programming has been used to address stochasticity in water and food management and include ecological quality in the decision-maker's utility function. Besides the direct integration of uncertainties into the decision-making process, dynamic programming has several other

advantages over alternative methods. One advantage is handling a long-time horizon (e.g., several decades with monthly intervals). A second advantage is that stochastic dynamic programming can use approximation methods to account for continuous decisions and state variables. In addition, stochastic dynamic programming is a flexible framework able to capture the optimal trade-offs and synergies in adaptation decision-making by modelling the risk preferences of a decision agent under uncertainties (Muccione et al., 2023).

For example, Muccione et al. (2023) used dynamic programming to present a stylised application of stochastic dynamic programming for local adaptation decision-making for a small alpine community exposed to debris flows and foods. Fig 4 depicts a schematic representation of a single period in the dynamic decision framework. Note that at the beginning of each period, the decision maker observes a unique state of the world, i.e. time, riverbed height historically maximum vulnerable height of house asset and the realisations of the two shocks related to foods**,** and debris flow. Given that unique state of the world, the decision maker considers the possible future stochastic occurrence of foods and debris flow and chooses the optimal level of excavation and if the options of dam building or relocation and the building of an alternative road should be executed.

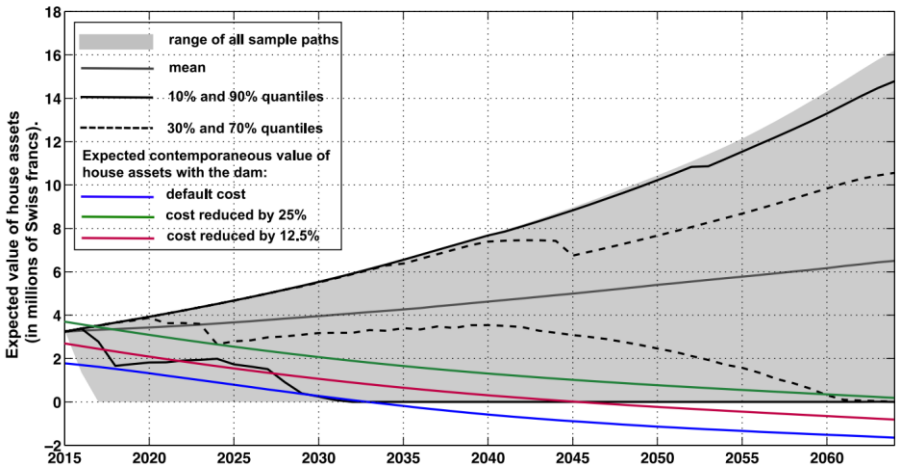
Fig 4: Time axis of decisions of example problem



Source:(Muccione et al., 2023)

This variable was transferred into the simulation, which simulated 10,000 randomised values. Then, calculations were conducted based on SDP (stochastic dynamic programming), and the results were statistics for the expected value of housing assets from 100,000 simulated paths and sensitivity regarding the cost of the dam. In addition, the expected value of the house assets is lower costs of building and maintaining the dam. All is shown in Fig.5 (Muccione et al., 2023), that is just for illustration of possible solution depiction.

Fig 5: Predicted development of the problem in future with the impact of decision maker



Source:(Muccione et al., 2023)

3 Results and discussion

We wanted to apply the methodology of dynamic programming in some sort of simple problem in financial management. This problem can be defined as the decision maker wants to maximise his profit from investing. The decision maker will make decisions is *k* years (total number of years), and he can make two decisions, he can withdraw some fraction of money or will let the stock money grow in value. There are however two main complications.

First is that the decision maker cannot withdraw all his money in the one go but he can withdraw just fraction of it. The second complication is that the investment project is risky and there is no guarantee that it is possible to withdraw money after all stages. We have chosen, substandard approach for financial management on how to calculate such problem, that is reproduction model (RM) that is widely used in environmental management.

In the RM literature, there are two models in which agents harvest a resource simultaneously. The first model focuses on situations where the resource users diminish the relative value per resource unit in the current period as their harvest level increases. However, the future value of the resource is undiminished. In contrast, in the dynamic RM, the resource's current users reduce the resource's level, thereby harming future users. Uncertainty of resource levels tends to promote over-harvesting, while resource scarcity induces greed. Thus, a more complete picture can be established by explicitly considering resource dynamics on the one hand and macroeconomic and social dynamics on the other.

There are many modifications to the RM. We even modified the RM for a specific problem unrelated to ecology, and the problem is an investment problem where the investor tries to make the most money possible from stocks. However, the investor can have three different approaches to strategies based on whether the investor prefers the money in cash or stock or does not mind, as long as the yield is at the maximum. To formulate such a problem, we must first declare sets and variables.

Description of the elements in Tab 4, 5, 6 and in formulas below are completely separated from previous notations of elements and formulas. Sets are described in Tab.4.

Tab. 4: Definition of sets

Symbol	Definition
$T=\{1,2,\ldots,k\}$	Stages (years of the investment horizon), where k is total number of years.
$I=\{1,2,\ldots,Z\}$	Admissible actions at each stage; *Z* is the number of distinct withdrawal policies that could be considered.

Source: Own Elaboration

A strategy over the whole horizon is an ordered *k*-tuple $(i_1,i_2,\ldots,i_k)$ with $i_t \in I$ for every *t*. The cartesian product I^k is simply the set of all such *k*-step strategies. Decision variables and auxiliary quantities are described in Tab.5.

Tab. 5: Definition of variables and auxiliary quantities

Symbol	Meaning (all evaluated at stage *t* unless stated otherwise)
N_t	Monetary value of the stock at the start of stage *t*.
π_{t,i_t}	Cash withdrawn in stage *t* when action i_t is chosen.
CASH	Total cash withdrawn over the entire horizon (objective).
STOCK	Final value of the remaining stock (secondary objective, if needed).

Source: Own Elaboration

Parameters are described in Tab.6.

Tab. 6: Definition of parameters

Symbol	Meaning
$\mathfrak{C}$	Growth (or depreciation) factor that multiplies the stock from one stage to the next.
a_{t,i_t}	Percentage of stock withdrawn if action i_t is taken at stage t.
A	Regulatory or self-imposed upper bound on the withdrawal percentage in any single stage.

Source: Own Elaboration

There are two objective functions, primary and secondary. Primary goal – maximise total cash withdrawals:

$$CASH = \max_{(i_1,\dots,i_k)\in I^k} \sum_{t=1}^{k} \pi_{t,i_t} \tag{13}$$

Optional secondary goal – maximise residual stock at the horizon

$$STOCK = N_{k+1} \tag{14}$$

where N_{k+1} is obtained from the recursion below.
Constraints :

1. Withdrawal-percentage limit

$$a_{t,i_t} \leq A \quad t \in T, i_t \in I \tag{15}$$

2. Cash actually withdrawn in stage t

$$\pi_{t,i_t} = a_{t,i_t}(N_t\mathfrak{C}) \quad t \in T, i_t \in I \tag{16}$$

3. Evolution of the stock

$$N_{t+1} = (N_t\mathfrak{C}) - \pi_{t,i_t} \qquad t = 1,\dots,k-1 \quad t \in T, i_t \in I \tag{17}$$

We used this model on the illustrative example of problem. The data is made up, however it reflects possible data from reality. We have calculated four different scenarios of this problem. The common values of variables were:

$N_1 = 100$
$k=3$
$Z=2$

For all illustrative scenarios, there are two possible actions Z: $(a_{t,1_t}; a_{t,2_t}) = (0; A)$.

The changing parameters in each of these problems were $\mathfrak{C}$ and A. In the Tab.7 are marked these four various combinations:

Tab. 7: Four different illustrative scenarios.

1. Combination: $\mathfrak{C}$=1,25; A=50%	2. Combination: $\mathfrak{C}$=1,2; A=50%
3. Combination: $\mathfrak{C}$=1,5; A=33%	4. Combination: $\mathfrak{C}$=1,35; A=80%

Source: Own Elaboration

We could use Excel and the add-on Solver to find the optimal solution for this model. This model is nonlinear, and for that matter we cannot use the simplex LP algorithm in the Solver, so we used the Generalized Reduced Gradient algorithm instead. The calculations of a problem with different input parameters, noted in Tab. 7, can be found in Tab. 8.

Decision-makers could have three approaches to this problem. First, he does not trust the investment project and only cares about cash, which he has at the end of the investment period, which has the most liquid form, approach *CASH*. The second approach is that the decisionmaker fully trusts the investment project and believes that the offered stock will have total value even after time *k*, and he wants to have as much money bound to those stocks as possible, that is approach *STOCK*. The third approach is that the decisionmaker believes in the investment project but not fully and wants to have as much money as possible at the end of time *k*; that approach is called *CASH+STOCK*.

Tab. 8: Results of Solver calculation of various combinations of problems parameters

	$\mathcal{C}$=1,25;A=50%	$\mathcal{C}$=1,2;A=50%	$\mathcal{C}$=1,5;A=33%	$\mathcal{C}$=1,35;A=80%
CASH	$(a_{1,i_1^*}, a_{2,i_2^*}, a_{3,i_3^*}) = (0,50,50)$ *CASH*=126.96	$(a_{1,i_1^*}, a_{2,i_2^*}, a_{3,i_3^*}) = (50,50,50)$ *CASH*=117.6	$(a_{1,i_1^*}, a_{2,i_2^*}, a_{3,i_3^*}) = (33,33,33)$ *CASH*=149.25	$(a_{1,i_1^*}, a_{2,i_2^*}, a_{3,i_3^*}) = (0,0,80)$ *CASH*=196.83
STOCK	$(a_{1,i_1^*}, a_{2,i_2^*}, a_{3,i_3^*}) = (0,0,0)$ *STOCK*=195.31	$(a_{1,i_1^*}, a_{2,i_2^*}, a_{3,i_3^*}) = (0,0,0)$ *STOCK*=172.8	$(a_{1,i_1^*}, a_{2,i_2^*}, a_{3,i_3^*}) = (0,0,0)$ *STOCK*=337.5	$(a_{1,i_1^*}, a_{2,i_2^*}, a_{3,i_3^*}) = (0,0,0)$ *STOCK*=245.04
CASH+ STOCK	$(a_{1,i_1^*}, a_{2,i_2^*}, a_{3,i_3^*}) = (0,0,0 \text{ or } 50)$ *CASH+ STOCK*=195.31	$(a_{1,i_1^*}, a_{2,i_2^*}, a_{3,i_3^*}) = (0,0,0 \text{ or } 50)$ *CASH+ STOCK*=172.8	$(a_{1,i_1^*}, a_{2,i_2^*}, a_{3,i_3^*}) = (0,0,0 \text{ or } 33)$ *CASH+ STOCK*=337.5	$(a_{1,i_1^*}, a_{2,i_2^*}, a_{3,i_3^*}) = (0,0,0 \text{ or } 80)$ *CASH+ STOCK*=245.04

Source: Own Elaboration

4 Conclusion

This article aimed to show how dynamic programming could be used to find the optimal use of processes in a simple problem. We believe that this feat was accomplished. We showed sample problems and the means of solving them. We even created a custom model that could be further extended to solve various economic or ecology-based problems. We used this method on a test problem, using manual calculation and even calculation based on the non-linear programming algorithm called Generalized Reduced Gradient.

The following research will focus on further developing our custom model. It will be developed to calculate various management optimisation problems using replication model inspired mathematical model, as we believe it could bring different views on optimising problems in dynamic and deterministic settings, leading to more accurate, faster results. It can also be used to merge or synchronise different management optimisation problems, such as environmental and financial management problems, leading to a more sustainable and profitable future.

The paper was elaborated in the scope of the grant assignment ESG I-24-111-00 ***Exploring the use of artificial intelligence to support the creation of scientific research papers and the data mining within them***

References

1. Abiy, T., Pang, H., & Williams, C. (2017). Dijkstra's Shortest Path Algorithm: Brilliant math & science wiki. Brilliant. https://brilliant.org/wiki/dijkstras-short-path-finder
2. Baeldung, W. (2023, March 24). Epsilon-Greedy Q-Learning. Baeldung on Computer Science. https://www.baeldung.com/cs/epsilon-greedy-q-learning
3. Bellman, R. (1954). *The theory of dynamic programming.* Bulletin of the American Mathematical Society, 60(6), 503-515. https://doi.org/10.1090/S0002-9904-1954-09848-8
4. Carlson, D. A., Haurie, A., & Leizarowitz, A. (1991). *Infinite-horizon optimal control: Deterministic & stochastic systems.* Springer. https://doi.org/10.1007/978-3-642-76755-5
5. Chebil, K., & Khemakhem, M. (2015). A dynamic programming algorithm for the Knapsack problem with setup. Computers & Operations Research, 64, 40–50. https://doi.org/10.1016/j.cor.2015.05.005
6. Davidsen, C., Pereira-Cardenal, S. J., Liu, S., Mo, X., Rosbjerg, D., & Bauer-Gottwein, P. (2015). *Using stochastic dynamic programming to support water resources management in the Ziya River Basin, China.* Journal of Water Resources Planning and Management, 141(7), 04014086. https://doi.org/10.1061/(ASCE)WR.1943-5452.0000482
7. Jaffar, J., Santosa, A. E., & Voicu, R. (2008). *Efficient memoisation for dynamic programming with ad-hoc constraints.* In Proceedings of AAAI-08 (pp. 297-302). https://doi.org/10.1609/aaai.v22i2.2190
8. Laščiak, A., Sojka, J., Šimkovic, J., Maňas, M., Hozlár, E., Unčovský, L., Chobot, M., Hušek, R., & Ulašin, V. (1983). *Optimálne programovanie*. Alfa.
9. Lu, Q., & Wang, Z. (2013). *A stochastic dynamic-programming approach for the equipment-replacement optimisation problem under utilisation uncertainty.* Transportation Research Procedia, 7, 400-409. https://doi.org/10.1016/j.trpro.2013.06.013
10. Muccione, V., Lontzek, T., Huggel, C., Ott, P., & Salzmann, N. (2023). An application of dynamic programming to local adaptation decision-making. Natural Hazards, 119(1), 523–544. https://doi.org/10.1007/s11069-023-06135-2
11. Munch, S. B., & Brias, A. (2024). *Empirical dynamic programming for model-free ecosystem-based management.* Methods in Ecology and Evolution, 15(4), 769–778. https://doi.org/10.1111/2041-210X.14302
12. Sieniutycz, S., & Jeżowski†, J. (2013). Dynamic Optimization Problems. Energy Optimization in Process Systems and Fuel Cells, 45–84. https://doi.org/10.1016/b978-0-08-098221-2.00002-3
13. So, M. K., & Kek, S. L. (2020). *Workforce-size problem in manufacturing with a dynamic-programming approach.* AIP Conference Proceedings, 2266, 090005. https://doi.org/10.1063/5.0018444
14. Sutton, R. S., & Barto, A. G. (2018). *Bellman equations and dynamic programming - university of Texas at Austin*. Introduction to Reinforcement Learning. https://login.cs.utexas.edu/sites/default/files/legacy_files/research/documents/6%20Bellman%20Eqs%20and%20DP.pdf
15. Taha, H. A. (2017). Operations research: An introduction. Pearson.
16. Wimmer, S., Hu, S., & Nipkow, T. (2018). *Verified memoisation and dynamic programming.* In Interactive Theorem Proving 2018 (LNCS 10895, pp. 579-596). https://doi.org/10.1007/978-3-319-94821-8_34

17. Yu, F., & Kuang, H. (2015). *Port multi-period investment-optimisation model based on supply-demand matching using dynamic programming.* Journal of Shipping & Trade, 2(4), 75-89. https://doi.org/10.1515/JSSI-2015-0077