Dynamic Programming Algorithm for Multiple Constraints Stock Portfolio Optimization

Manuel Thimoty Silalahi
NIM : 13524102
Program Studi : Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, JL. Ganesha 10 Bandung
13524102.std.stei.itb.ac.id, manuelsilalahi06@gmail.com

***Abstract*—Traditional portfolio optimization models frequently rely on historical price data and linear constraints, often overlooking qualitative market dynamics and the discrete nature of real-world stock trading. While heuristic approaches, such as greedy algorithms, seek optimal solutions at every local stage, they frequently fail to guarantee global optimality. Conversely, exhaustive recursive methods that iterate through all possible solutions suffer from severe computational bottlenecks, making them impractical for large-scale financial markets. To bridge this gap, integrating mathematical finance with dynamic programming (DP) provides a rigorous framework. DP efficiently tackles multi-stage optimization by breaking down complex problems and storing sub-problem results to prevent redundant calculations. However, while DP offers exact solutions for knapsack-based discrete asset allocation, expanding the decision criteria to multi-dimensional constraints typically triggers a combinatorial explosion, widely known as the curse of dimensionality. To overcome this limitation, this paper proposes a novel, multi-dimensional DP algorithm that optimizes integer-based stock selection by tracking exactly three transition states: available capital budget, cumulative portfolio risk, and liquidty . Furthermore, to maintain polynomial-time computational feasibility across the expanded state space, a heuristic state-space pruning mechanism is introduced, effectively bounding the search tree and eliminating suboptimal allocation paths.**

I. Introduction

A. Background

Portfolio optimization is a fundamental problem in quantitative finance, central to maximizing investment returns while adhering to specific investor constraints. While traditional continuous models, such as Markowitz's Modern Portfolio Theory, assume assets are infinitely divisible, real-world financial markets mandate trading in strictly discrete units or "lots." This discrete nature, combined with the necessity to satisfy a multidimensional constraint environment fundamentally transforms the continuous optimization problem into a variation of the bounded multi-dimensional knapsack problem, which is strictly NP-hard.

Solving this multi-constrained discrete allocation problem presents a formidable computational challenge. Heuristic algorithms frequently fail to guarantee a globally optimal portfolio, while naive exact methods like exhaustive search suffer from exponential time complexity ($\mathcal{O}(K^N)$). Dynamic Programming (DP) offers a mathematically rigorous alternative by leveraging optimal substructure. However, integrating multiple independent constraints forces the state space to expand multiplicatively, leading to the curse of dimensionality and severe memory limitations. To address this gap, this paper proposes an optimized 3-state Dynamic Programming model integrated with a heuristic state-space pruning mechanism. By strictly evaluating forward-reachable active states, the algorithm circumvents combinatorial explosion, ensuring that multi-constraint global portfolio optimization remains both mathematically exact and computationally feasible within standard memory limits.

B. Problem Formulation

The core problem addressed in this research is the optimal discrete allocation of finite capital across a universe of N candidate stocks to maximize the portfolio's total expected return. Computationally, this scenario translates to a Bounded Multidimensional Knapsack Problem (MDKP). The algorithm must determine the exact integer vector of lot allocations $\mathbf{x} = \{x_1, x_2, \ldots, x_N\}$, subject to specific lower and upper bound purchasing limits for each asset. Furthermore, this maximization objective is strictly governed by three intersecting global constraints: the total purchasing cost (including transaction fees) must not exceed a predefined capital budget (B), the aggregated portfolio volatility must stay below a maximum risk tolerance ($R_{\max}$), and the cumulative asset liquidity must meet or surpass a minimum required threshold ($L_{\min}$).

The primary algorithmic challenge lies in navigating the combinatorial explosion inherent in this multidimensional constraint environment. Constructing a standard Dynamic Programming table requires tracking a 3-dimensional state space ($B \times R_{\max} \times L_{\min}$), which rapidly becomes computationally intractable for large boundaries. Therefore, the problem extends beyond establishing the mathematical objective function; it necessitates the design of an optimized state-transition mechanism that can systematically isolate and evaluate only feasible allocation paths, ensuring the discovery of the globally optimal portfolio while operating strictly within practical memory and time complexities.

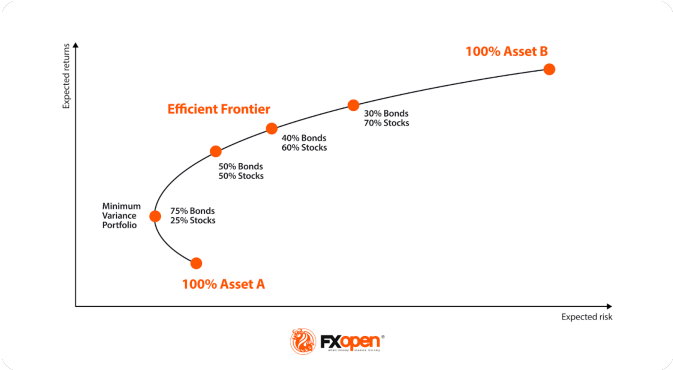

Fig. 1. MPT

C. Objectives

Based on the formulated problem, this study outlines the following primary objectives:

1) To design and formulate a multi-dimensional Dynamic Programming algorithm capable of finding the globally optimal integer-based asset allocation under strict capital, risk, and liquidity constraints.
2) To introduce and implement a heuristic state-space pruning mechanism to significantly reduce the spatial and temporal complexity, effectively bypassing the curse of dimensionality.
3) To empirically validate the accuracy and computational efficiency of the proposed algorithm using real-world historical stock data from the Indonesian Stock Exchange (IDX), analyzing the constraint-binding behavior and the resulting optimal portfolio composition.

II. Theoretical Framework

A. Modern Portfolio Theory and Discrete Allocation

The foundational mathematical framework for asset allocation is Modern Portfolio Theory (MPT), introduced by Harry Markowitz in 1952 [1]. MPT revolutionized financial economics by formalizing the concept of diversification, positing that rational investors seek to construct an efficient frontier that maximizes expected returns for a given level of variance (risk). In its classical formulation, MPT models the portfolio selection as a continuous convex optimization problem, where the decision variables—representing the proportion of capital allocated to each asset—are strictly continuous [2]. This implies an underlying assumption that financial assets are infinitely divisible and can be traded in exact fractional shares.

However, the practical implementation of MPT in real-world equity markets, such as the Indonesian Stock Exchange (IDX), directly contradicts this continuous assumption. Market regulations strictly mandate that equities be traded in discrete, standardized integer units known as ”lots” (e.g., 1 lot equates to 100 shares in the IDX). Consequently, the continuous optimization model becomes practically invalid. When continuous asset weights are forcibly rounded to the nearest allowable integer lot, the resulting portfolio frequently violates stringent capital or risk constraints, or strays significantly from the global optimum [3]. This fundamental limitation necessitates the reformulation of the portfolio selection model from a continuous mean-variance problem into a discrete combinatorial optimization framework, mathematically reflecting the rigid integer constraints of real-world trading.

B. The Multidimensional Bounded Knapsack Problem

The transition from continuous asset weights to discrete share allocations mathematically maps the portfolio optimization problem into the domain of combinatorial optimization, specifically adapting the classic Knapsack Problem (KP) [4]. In its canonical form, the KP seeks to select a subset of items—each defined by a specific value and weight—such that the total value is maximized without exceeding the knapsack's capacity. In the context of equity trading, the candidate ”items” correspond to the available stocks, the ”values” denote the expected returns per lot, and the primary ”capacity” represents the investor's total capital budget.

However, the real-world trading environment imposes additional operational complexities that extend this canonical model into a Bounded Multidimensional Knapsack Problem (MDKP) [5]. The problem is formally ”bounded” because the selection of each asset is restricted by specific lower and upper lot limits (l_i and u_i) governed by the investor's diversification strategy. Furthermore, the problem is ”multidimensional” because the optimization is constrained not merely by a single financial budget, but by multiple independent resource thresholds. In this study, these dimensions correspond to the strict upper bound on risk tolerance and the strict lower bound on portfolio liquidity.

The MDKP is a well-documented strongly NP-hard problem [6]. As the number of candidate assets and the dimensions of the constraints increase, the feasible combinatorial solution space expands exponentially. Formally, if we use N different stocks and K different boundaries or metrics that is used for the computation, we have to define at least $N \cdot K$ state. Consequently, approximate heuristics and greedy algorithms (which typically rely on sorting items by simple value-to-weight ratios) frequently fail to satisfy the multi-dimensional constraint intersections and cannot guarantee a globally optimal solution. This inherent computational intractability necessitates the deployment of exact combinatorial methodologies, establishing the foundation for utilizing advanced Dynamic Programming to systematically navigate the constraint boundaries and derive the optimal discrete allocation.

C. Dynamic Programming Paradigm

To exactly solve the NP-hard multidimensional knapsack problem without falling into the trap of exponential time complexity, this study employs the Dynamic Programming (DP) paradigm. As widely documented in competitive programming literature [8], DP is an algorithmic optimization technique that systematically solves complex computational problems by breaking them down into simpler, smaller sub-problems.

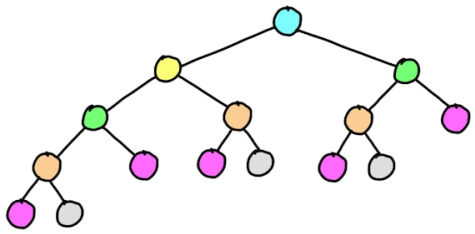

Fig. 2. Dynamic Programming Visualization

The efficiency of DP relies entirely on two fundamental problem properties. First, the problem must exhibit an *optimal substructure*, meaning that the global optimal solution can be constructed directly from the optimal solutions of its subproblems. In the context of discrete portfolio selection, the optimal allocation of i assets is built upon the optimal allocation already computed for $i-1$ assets. Second, the problem must contain *overlapping subproblems*. A naive exhaustive recursive search would repeatedly evaluate the exact same budget and risk combinations millions of times. DP overcomes this by calculating each sub-problem exactly once and storing the result in a state table—a technique known as tabulation (bottom-up DP).

By evaluating state transitions iteratively and updating this table, the DP algorithm guarantees the discovery of the absolute global optimal return for discrete allocations. It bypasses the inaccuracies inherent in greedy heuristics while drastically reducing the execution time compared to brute-force combinatorial search.

D. The Curse of Dimensionality and State-Space Pruning

While Dynamic Programming effectively resolves the overlapping subproblems of discrete allocation, expanding the decision criteria introduces a severe computational bottleneck coined by Richard Bellman as the curse of dimensionality [9]. In a standard multidimensional DP approach, the state space is represented as a dense multi-dimensional hypergrid. For a portfolio problem constrained by budget (B), maximum risk ($R_{\max}$), and minimum liquidity ($L_{\min}$), the algorithm must theoretically allocate and iterate through $(B+1) \times (R_{\max}+1) \times (L_{\min}+1)$ memory cells for every candidate asset. As these thresholds scale to realistically accommodate financial market data, this multiplicative expansion rapidly consumes available memory and processing capacity, invariably leading to computational exhaustion such as Memory Limit Exceeded (MLE) and Time Limit Exceeded (TLE) during execution.

To overcome this inherent limitation without sacrificing mathematical exactness, this study integrates a heuristic state-space pruning mechanism, often referred to as Forward-Reachable or Sparse DP. Rather than exhaustively traversing the entire dense grid at each stage, the algorithm maintains a dynamic list of active states—specific coordinates within the state space that have been genuinely reached via valid allocation paths. During state transitions, the algorithm exclusively evaluates and branches outward from these active states. By systematically ignoring the vast regions of the multidimensional grid that remain unreachable or unfeasible, this pruning strategy effectively bounds the search tree. This ensures that the multidimensional knapsack model remains computationally performant and feasible within standard operational memory limits.

III. Problem Formulation and Mathematical Modelling

A. Mathematical Formulation

Here, the mathematical portfolio optimization will be modeled as a multi constraint, integer based allocation. Let N be the total number of available assets in the market. Our main objective is to calculate the optimal number of each asset, such that the total expected return is maximized. We'll use 3 dimensional (strict) constraint : capital budget, risk tolerance, and minimum liquidity requirements. For the details, here are the notations and parameters that will be used :

- B: Total available capital budget.
- R_{max}: Maximum acceptable threshold for portfolio risk.
- L_{min}: Minimum required threshold for portfolio liquidity.
- $\mathcal{R}_i$: Expected return per unit of share i.
- ρ_i: Price per unit of share i.
- σ_i: Risk contribution per unit of share i (e.g., historical variance or Value-at-Risk).
- ν_i: Liquidity metric per unit of share i (e.g., Average Daily Trading Volume).
- τ_i: Transaction cost coefficient for asset i (representing broker fees, levies, and taxes as a percentage).
- l_i: Lower bound of shares if asset i is selected.u_i: Upper bound of shares if asset i is selected.

Note that we'll also use two sets of decisions variable. It's because we need to capture the discrete nature of stock trading.

- x_i: An integer variable representing the exact number of shares of asset i to be purchased.
- y_i: A binary indicator variable, where $y_i = 1$ if asset i is included in the portfolio, and $y_i = 0$ otherwise.

From this, we can concluded that our main objective is to maximize

$$Z = \sum_{i=1}^{N} \mathcal{R}_i x_i$$

that is the expected return value. But, note that this objection function is actually bounded by the following state and domain constraints :

1) Capital Budget Constraint (with Transaction Costs): The total cost of the purchased shares, inclusive of all applicable transaction fees, must not exceed the available capital.

$$\sum_{i=1}^{N} \rho_i x_i (1+\tau_i) \leq B$$

2) Risk Constraint: The aggregated risk of the selected assets must be strictly less than or equal to the maximum risk tolerance defined by the investor.

$$\sum_{i=1}^{N} \sigma_i x_i \leq R_{max}$$

3) Liquidity Constraint: The overall liquidity of the portfolio must meet or exceed the minimum threshold to ensure position mobility without severe slippage.

$$\sum_{i=1}^{N} \nu_i x_i \geq L_{min}$$

4) Quantity and Bounding Constraints: The number of shares purchased must fall within the predetermined minimum and maximum bounds, conditioned on the selection of the asset.

$$l_i y_i \leq x_i \leq u_i y_i, \quad \forall i \in \{1, 2, \ldots, N\}$$

5) Domain Constraints:

$$y_i \in \{0, 1\}, \quad \forall i \in \{1, 2, \ldots, N\}$$
$$x_i \in \mathbb{Z}^+, \quad \forall i \in \{1, 2, \ldots, N\}$$

B. Dynamic Programming Translation

Now, we'll models the mathematical problems and solve it using programming. One of the approaches can be done via recursive that uses exhaustive search to explore all possible combination that satisfies the constraint given. Unfortunately, this mechanism fail due to overlapping sub-problems across the vast search space. By discretizing the continuous constraints into specific grid intervals, the DP algorithm capitalizes on the optimal substructure of the portfolio problem, evaluating each asset sequentially while memoizing the optimal solutions for every combination of states.

We'll use 3 state representation parameters traversing across N stages. We define the maximum achievable expected return from a subset of assets from index i to N using the state function $f(i, b, r, l)$, where:

- i: The current asset index being evaluated ($1 \leq i \leq N$).
- b: The remaining available capital budget.
- r: The remaining risk tolerance capacity.
- l: The accumulated liquidity.

Thus, $f(i, b, r, l)$ denotes the optimal return from the sub-array of assets $[i \ldots N]$ given the current state parameters b, r, and l.

To ensure valid algorithmic termination and prevent out-of-bound memory access, the recurrence strictly adheres to the following base cases:

- Successful allocation: When the algorithm has evaluated all N assets, it returns 0 if the minimum liquidity threshold has been met, or $-\infty$ (a heavily penalized value) if the liquidity constraint fails.

$$f(N+1, b, r, l) = \begin{cases} 0, & \text{if } l \geq L_{min} \\ -\infty, & \text{otherwise} \end{cases}$$

- Budget or Risk exhaustion: If at any recursive depth the remaining budget b or risk tolerance r drops below zero, the path is invalidated:

$$f(i, b, r, l) = -\infty, \quad \text{if } b < 0 \text{ or } r < 0$$

At each stage i, the algorithm evaluates the optimal decision matrix. For a given asset i, the decision variable x_i (representing the number of shares) must bounded by l_i and u_i. The algorithm computes the maximum return by comparing two primary branches:

- Skip ($y_i = 0$): The algorithm chooses not to invest in asset i. The integer decision is $x_i = 0$. The state parameters transition to the next asset without any deduction or addition.

$$Value_{skip} = f(i + 1, b, r, l)$$

- Take ($y_i = 1$): The algorithm chooses to invest in asset i by selecting an optimal lot size $x_i \in [l_i, u_i]$. This action deducts the inclusive purchasing cost from the budget, deducts the specific risk from the risk capacity, and adds the asset's liquidity to the accumulated total.

$$Value_{take} = x$$

where $x = \max_{x_i \in [l_i, u_i]} \Big[\mathcal{R}_i x_i + f\big(i + 1, b - \rho_i x_i (1 + \tau_i), r - \sigma_i x_i, l + \nu_i x_i\big)\Big]$.

The global optimal substructure dictates that $f(i, b, r, l)$ will simply take the maximum value between skipping or taking the asset under valid constraints:

$$f(i, b, r, l) = \max\Big(Value_{skip}, Value_{take}\Big)$$

To initiate the optimization process and find the global optimal portfolio return, the algorithm evaluates $f(1, B, R_{max}, 0)$. By systematically filling the 3-state multidimensional array, the algorithm avoids recalculating overlapping allocation scenarios, maintaining mathematical exactness while determining the ideal distribution of shares.

C. Feasibility-Based State-Space Pruning

Although the worst-case theoretical time complexity of the proposed 3-state Dynamic Programming model remains bounded by the dimensions of its multidimensional state space, the practical execution time can be significantly accelerated by eliminating invalid allocation paths early in the recursion tree. To prevent the algorithm from exhaustively exploring combinatorial branches that will inevitably violate the constraints, we introduce a strict feasibility pruning mechanism.

The pruning heuristic evaluates whether the remaining unvisited assets possess the theoretical capacity to satisfy the portfolio's minimum liquidity constraint (L_{min}). To achieve $O(1)$ time complexity for this look-ahead validation during the state transitions, we utilize a precomputed suffix array. Let $MaxLiq_i$ denote the absolute maximum liquidity that can be accumulated from the remaining assets from index i to N. This value assumes the algorithm purchases the maximum

allowable bounds (u_j) for all remaining assets, precomputed in linear $O(N)$ time as follows:

$$MaxLiq_i = \sum_{j=i}^{N} (u_j \cdot \nu_j)$$

During the recursive state evaluation at asset i with current accumulated liquidity l, the algorithm triggers the feasibility bounding function. If the current liquidity combined with the maximum potential future liquidity is strictly less than the required threshold, the branch is entirely discarded:

$$\text{If } l + MaxLiq_i < L_{min}, \text{ then } f(i, b, r, l) = -\infty$$

By applying this condition, the algorithm mathematically guarantees that no computational resources are wasted on exploring sub-trees where fulfilling the liquidity constraint is fundamentally impossible. Furthermore, similar look-ahead validations are inherently enforced for the budget (b) and risk (r) constraints; transitions that immediately result in negative residual budget or negative risk tolerance are pruned without subsequent recursive calls.

IV. Algorithm Design and Implementation

To resolve the multidimensional knapsack model established in Section 2, we implement an iterative Bottom-Up Dynamic Programming (DP) algorithm. Compared to a top-down recursive approach, the procedural method completely eliminates function call overhead and recursion depth limits, providing superior execution speed and cache locality for large-scale financial datasets.

The core optimization maintains a 3-dimensional array tracking the available budget, cumulative risk, and achieved liquidity. To prevent unbounded memory growth on the liquidity state, any accumulated liquidity exceeding the required threshold is algorithmically capped at L_{min}. Furthermore, to reconstruct the actual portfolio allocation rather than just obtaining the maximum numerical return, an auxiliary tracking matrix (Choice) is synchronized during the state transitions to record the optimal lot multiplier x chosen at each stage. Also, note that this is only the pseudocode that demonstrates how the state of the dp will be. For the detail implementation, you can look at the repo. This involves some prunning strategies to avoid both Memory Limit Exceeded (MLE) and Time Limit Exceeded (TLE).

Algorithm 1 DP Transition

Require: Assets $A[1 \dots N]$, Budget B, Max Risk R, Min Liq L

Initialize $DP[0 \dots B][0 \dots R][0 \dots L]$ with $-\infty$
Initialize $Choice[1 \dots N][0 \dots B][0 \dots R][0 \dots L]$ with 0
$DP[0][0][0] \leftarrow 0$
for $i \leftarrow 1$ **to** N **do**
  $NextDP \leftarrow DP$
  for each state (b, r, l) **where** $DP[b][r][l] \neq -\infty$ **do**
    for $x \leftarrow A[i].min\_lot$ **to** $A[i].max\_lot$ **do**
      $Cost \leftarrow A[i].price \times x \times (1 + A[i].trans\_cost)$
      $(b', r') \leftarrow (b + Cost, r + A[i].risk \times x)$
      $l' \leftarrow \min(L, l + A[i].liquidity \times x)$
      if $b' \leq B$ **and** $r' \leq R$ **then**
        if $DP[b][r][l] + A[i].return \times x > NextDP[b'][r'][l']$ **then**
          $NextDP[b'][r'][l'] \leftarrow DP[b][r][l] + A[i].return \times x$
          $Choice[i][b'][r'][l'] \leftarrow x$
        end if
      end if
    end for
  end for
  $DP \leftarrow NextDP$
end for
return $\max(DP[b][r][L])$ **and** backtrack via $Choice$ for optimal allocations

V. EXPERIMENT AND RESULTS

A. Dataset Gathering

We'll use dataset that is available on kaggle. The datasets can be viewed here. The dataset contains 3 folders and 1 csv file that describes each stocks. The 3 additional folders contain each stock data over a time frame. For example, a folder named 'daily' contains csv file for every stock. Each CSV contains information about the stock every day, while a folder named 'month' will contain information about the stock every month. For example, table below represents the first five row in the daily/ACST.csv.

timestamp	open	low	high	close	volume
2013-06-24	2479	2384	2884	2694	3
2013-06-25	2741	2646	2884	2765	13010758
2013-06-26	2789	2718	2813	2741	25307989
2013-06-27	2741	2694	3147	3075	3
2013-06-28	3099	3075	3314	3099	9397503

To construct our experimental portfolio, we uniformly sampled $N = 35$ random stocks from the dataset using a constant random seed to serve as our candidate asset universe. The selected ticker symbols are listed in this table below.

PPGL	LPLI	GDYR	OBMD	BVIC
SKLT	WOMF	BBMD	ENZO	UNTR
TIRA	BOBA	ESIP	DAYA	AMAR
ISAT	EAST	ENVY	TRST	NICK
MDKI	ASRI	CTBN	RMKE	DSFI
LMPI	BHIT	MMLP	STAA	IDPR
ITIC	INCI	BOLA	INPS	APEX

B. Experimental Setup

The proposed multi-dimensional dynamic programming algorithm was implemented in C++17 and compiled with GCC under the `-O2` optimization flag. The experiment was conducted on a dataset comprising $N = 35$ actively traded stocks on the Indonesian Stock Exchange (IDX), loaded from a structured CSV file (`dataset-stima.csv`). The pre-processing parts is just simply taking random rows using constant seeding

value (for this, we'll use seed = 42 in the python). Each asset i is characterized by the tuple $(\rho_i, R_i, \sigma_i, \nu_i, \tau_i, l_i, u_i)$, representing the price per share, expected return, risk contribution, liquidity score, transaction cost coefficient, and lower/upper lot bounds, respectively.

The baseline experiment was configured with the following constraint thresholds:

- Capital Budget: $B = 100$ (scaled integer units).
- Maximum Risk Tolerance: $R_{\max} = 1500$.
- Minimum Liquidity Requirement: $L_{\min} = 200$.

All monetary and risk parameters were pre-scaled to integer-compatible values to ensure compatibility with the discrete state-space representation.

C. Implementation and Pruning Strategy

The algorithm follows an iterative bottom-up DP formulation with a 3-dimensional state tuple (b, r, l), denoting the consumed capital, accumulated risk, and accumulated liquidity, respectively. To circumvent the curse of dimensionality inherent in dense 3D grid traversal, a heuristic state-space pruning mechanism was employed. Rather than exhaustively evaluating every cell in the $(B+1) \times (R_{\max}+1) \times (L_{\min}+1)$ hypergrid at each stage, the algorithm maintains a sparse list of `active_states`. This list exclusively tracks states that were actually reached in the preceding stage. Consequently, transitions are only computed from feasible states, effectively bounding the branching factor and eliminating redundant exploration of unreachable regions in the state space.

At each stage i, for every active state and for every feasible lot size $x_i \in [l_i, u_i]$, the algorithm computes:

$$\text{cost} = \rho_i x_i (1 + \tau_i), \tag{1}$$

$$\text{risk} = \sigma_i x_i, \tag{2}$$

$$\text{liquidity} = \nu_i x_i. \tag{3}$$

that is cost, risk, and liquidity when taking stock i with amount of x_i. A transition is accepted only if the new state satisfies $\sum \text{cost} \leq B$ and $\sum \text{risk} \leq R_{\max}$. The expected return is maximized via the recurrence:

$$dp(b', r', l') = \max\left(dp(b', r', l'),\ dp(b, r, l) + R_i x_i\right). \tag{4}$$

D. Baseline Optimization Results

Table I presents the optimal portfolio allocation obtained from executing the DP algorithm over the full universe of 35 assets under the specified constraints.

TABLE I
OPTIMAL PORTFOLIO ALLOCATION

Metric	Value
Maximum Expected Return (Z)	0.02
6Budget Consumed	51 / 100
Total Risk	1496 / 1500
Asset Allocation	
RMKE	3 lots
BBMD	1 lot

The optimizer selected only two assets out of the 35 available candidates. The portfolio achieves a total expected return of 0.02 while strictly adhering to the imposed constraints. Notably, the risk capacity is saturated to 99.73% $(1496/1500)$, whereas only 51% of the available budget is utilized.

E. Discussion

1) Constraint Binding Analysis: The experimental results reveal that **risk tolerance** is the active bottleneck in this configuration. The algorithm exhausts nearly the entire risk budget (1496 out of 1500), leaving an insufficient margin (4 units) to accommodate any additional lot of either selected asset:

- Adding one more lot of RMKE would incur an additional risk of 438, exceeding the threshold.
- Adding one more lot of BBMD would incur an additional risk of 182, also exceeding the threshold.

In contrast, the budget constraint is non-binding $(51/100)$, indicating that the investor's capital is not the limiting factor. The liquidity requirement ($L_{\min} = 200$) is satisfied trivially; RMKE alone contributes $3 \times 3437 = 10311$ liquidity units, far surpassing the minimum threshold. Thus, liquidity does not restrict the solution space in this particular scenario.

2) Portfolio Composition Rationale: The selection of RMKE as the primary constituent is driven by its superior risk-adjusted return profile. With a per-unit expected return of 0.005023, allocating 3 lots contributes approximately 0.0151 to the total objective value. BBMD, while offering a lower per-unit return (0.000422), is included because its incremental risk cost (182) is sufficiently low to fit within the remaining risk capacity after RMKE's allocation, thereby providing a marginal gain without violating the hard risk limit. All remaining 33 assets were excluded because their inclusion would either yield a lower objective value or violate one or more constraints.

3) State-Space Efficiency: The effectiveness of the pruning strategy is highlighted by comparing the number of actively explored states against the theoretical maximum. The raw state-space volume is:

$$(B+1) \times (R_{\max}+1) \times (L_{\min}+1) = 101 \times 1501 \times 201 = 30.4 \times 10^6 \text{cells}$$

In a naive dense DP implementation, every stage would require iterating over all 30.4 million cells, leading to prohibitive computational costs. However, by restricting transitions some active states the algorithm explored only a minute fraction of this space. This sparsity ensures that the effective complexity remains polynomial in the number of assets and the constraint bounds, rather than exploding combinatorially.

F. Summary

The experiments demonstrate that the proposed 3-state DP algorithm successfully derives the globally optimal integer portfolio allocation under multi-dimensional constraints. The risk constraint emerged as the dominant limiting factor, while the budget and liquidity constraints were non-binding. The heuristic state-space pruning mechanism proved essential in maintaining computational tractability, validating the algorithm's feasibility for discrete portfolio optimization problems.

LINKS

The project repository can be viewed here
The dataset source (Kaggle) can be viewed here here
Short explanation can be viewed here

REFERENCES

[1] H. Markowitz, "Portfolio Selection," *The Journal of Finance*, vol. 7, no. 1, pp. 77–91, 1952.
[2] G. Cornuejols and R. Tütüncü, *Optimization Methods in Finance*. Cambridge, UK: Cambridge University Press, 2006.
[3] R. Mansini and M. G. Speranza, "Heuristic algorithms for the portfolio selection problem with minimum transaction lots," *European Journal of Operational Research*, vol. 114, no. 2, pp. 219–233, 1999.
[4] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems*. Berlin, Heidelberg: Springer-Verlag, 2004.
[5] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*. Chichester, UK: John Wiley & Sons, 1990.
[6] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY: W. H. Freeman and Company, 1979.
[7] F. V. Jezeie, S. J. Sadjadi, and A. Makui, "Constrained portfolio optimization with discrete variables: An algorithmic method based on dynamic programming," *PLOS ONE*, vol. 17, no. 7, p. e0271811, Jul. 2022.
[8] CP-Algorithms, "Dynamic Programming - Knapsack," *cp-algorithms.com*. [Online]. Available: https://cp-algorithms.com/dynamic_programming/knapsack.html.
[9] R. E. Bellman, *Dynamic Programming*. Princeton, NJ: Princeton University Press, 1957.

VI. PERNYATAAN

Dengan ini saya mengatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung. 19 Juni 2026

Manuel Thimoty Silalahi