Applying Dynamic Programming to the Multiple-Choice Knapsack Problem for Optimizing Armor Loadouts in Elden Ring

Stevanus Agustaf Wongso – 13524020
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Bandung, Indonesia
13524020@std.stei.itb.ac.id
stevanus610@gmail.com

***Abstract*—In Elden Ring, a player equips one armor piece in each of four slots, and the combined weight must stay within a fraction of the Equip Load the player can carry 30% for a light roll, 70% for a medium roll, or 100% for a heavy roll. Players pursue different goals, maximizing physical, magic, or fire damage negation, a status resistance, or poise. Choosing the combination that maximizes a chosen attribute under the weight limit is a combinatorial optimization problem, which players usually solve by greedy. This paper models the task as the Multiple-Choice Knapsack Problem and solves it exactly with dynamic programming. We implement a program that, given an armor dataset, any target attribute, and a weight budget, returns the optimal loadout, removing the manual guesswork players normally face. To assess the benefit, we compare the optimal loadout against the greedy approach across a range of Equip Load budgets on a dataset of 568 armor pieces; for physical damage negation, the greedy method falls short of the optimum by up to 65.1%, showing that manual selection can leave meaningful value unused.**

***Index Terms*—dynamic programming, multiple-choice knapsack, combinatorial optimization, greedy heuristic, equip load, Elden Ring**

I. Introduction

Elden Ring is an open-world action role-playing game developed by FromSoftware and published by Bandai Namco Entertainment, directed by Hidetaka Miyazaki with world-building by the fantasy author George R. R. Martin, and released on 25 February 2022. The player controls a customizable character, the Tarnished, who explores a vast world called the Lands Between, defeats powerful bosses, and grows stronger by spending an in-game currency called runes to raise attributes and by equipping weapons, armor, talismans, and spells. The game belongs to the "Soulslike" genre that FromSoftware established through the Dark Souls series, Bloodborne, and Sekiro, a genre known for high difficulty and deep, highly customizable character builds. Commercially, the game was an immediate and lasting success. It sold around 12 million copies within its first weeks and 13.4 million by the end of March 2022, and surpassed 20 million before its first anniversary [1]. By April 2025 it had reached 30 million copies, becoming the fastest-selling game in Bandai Namco's history and one of the best-selling video games of all time [1]. Critically it was equally dominant, earning more than 300 Game of the Year awards and winning the top Game of the Year honor at The Game Awards, the D.I.C.E. Awards, and the Golden Joystick Awards [1]. The game's momentum has not faded. Its expansion, Shadow of the Erdtree, released in June 2024, sold five million copies within three days and surpassed ten million by July 2025 [2], and a standalone spin-off, Elden Ring Nightreign, followed in 2025. Beyond sales, the title commands an enormous online community: at launch it topped streaming charts with roughly 125 million hours watched on Twitch in a single month [1]. This large, highly engaged player base continuously discusses strategies, character builds, and equipment choices. Much of that engagement centers on the game's deep build system. Because every weapon, spell, and armor piece interacts with the character's stats and a strict weight limit known as Equip Load, players invest considerable effort in optimizing their loadouts, and an entire ecosystem of community-made build planners and calculators has emerged to support them. One recurring decision is the choice of armor: selecting the combination of pieces that maximizes a desired protective attribute while staying within the weight the character can carry. Players usually make this choice manually, by intuition or trial and error, with no guarantee that the result is optimal. This paper formalizes that decision as a combinatorial optimization problem and solves it exactly, providing a tool that removes the guesswork.

II. Theoretical Foundation

A. The Knapsack Problem

The 0/1 (integer) knapsack problem is a classic combinatorial optimization problem and is well known to be NP-hard [6]. Given a knapsack of weight capacity M and n objects, where each object i has a weight w_i and a profit p_i, the goal is to choose a subset of objects that maximizes the total profit without exceeding the capacity [4]. Formally,

$$\text{maximize } F = \sum_{i=1}^{n} p_i x_i \tag{1}$$

$$\text{subject to } \sum_{i=1}^{n} w_i x_i \le M, \quad x_i \in \{0,1\}, \tag{2}$$

where $x_i = 1$ if object i is selected and $x_i = 0$ otherwise [4]. Because each object is either fully taken or left out, the number of candidate subsets grows as 2^n, so an exhaustive search becomes impractical as n increases. This is what makes a structured method such as dynamic programming valuable.

B. The Multiple-Choice Knapsack Problem (MCKP)

The Multiple-Choice Knapsack Problem generalizes the 0/1 knapsack problem by partitioning the objects into classes and requiring that exactly one object be chosen from each class, still subject to the weight capacity [6]. This structure matches armor selection directly: each equipment slot : head, chest, arms, and legs, is a group, and the player equips exactly one piece per slot. The grouping does not remove the combinatorial difficulty; If each of m groups offers t candidates, there are t^m possible loadouts, which again motivates an exact, efficient method.

C. Dynamic Programming

Dynamic programming is a method that solves a problem by decomposing its solution into a sequence of stages, so that the overall solution can be viewed as a series of interrelated decisions [4]. The term "dynamic" refers to the use of a table that is gradually filled during the computation, while "program" here means planning rather than coding [4].

The method rests on the Principle of Optimality: if the overall solution is optimal, then the partial solution up to any stage k is also optimal [4]. In practice this means the computation can advance from stage k to stage $k+1$ using the optimal result already obtained at stage k, without recomputing earlier stages, an instance of optimal substructure combined with overlapping subproblems, where the same partial result is reused many times instead of being recalculated. A problem is suitable for dynamic programming when it can be divided into stages with a single decision taken at each stage, and when each stage consists of a number of states representing the possible inputs to that stage [4]. The solution can proceed in a forward direction, from stage 1 to n, or in a backward direction, from stage n to 1 [4].

Developing such an algorithm follows four steps: characterize the structure of the optimal solution, define the optimal value recursively, compute it forward or backward using a table, and optionally reconstruct the optimal solution [4]. Unlike a greedy approach, which commits to only one sequence of decisions, dynamic programming considers more than one sequence and can therefore guarantee the global optimum [4].

Applied to the knapsack problem, let stage k be the decision of whether to include object k, and let the state y be the remaining capacity. Writing $f_k(y)$ for the maximum profit attainable at stage k with capacity y, the recurrence is

$$f_0(y) = 0,\ y \ge 0; \qquad f_k(y) = -\infty,\ y < 0, \tag{3}$$

$$f_k(y) = \max\{\, f_{k-1}(y),\ p_k + f_{k-1}(y - w_k)\,\}, \quad k = 1, \ldots, n, \tag{4}$$

and the optimal value is $f_n(M)$ [4]. At each step the method compares leaving object k out, $f_{k-1}(y)$, against taking it, $p_k + f_{k-1}(y - w_k)$, and keeps the larger [4]. The table has n stages and $M + 1$ capacity states, and each entry is computed in constant time, giving a time and space complexity of $O(nM)$. This is pseudo-polynomial: efficient when M is moderate, as it is for the armor weight budgets considered here.

D. Greedy Heuristics

A greedy algorithm constructs a solution through a sequence of locally optimal choices and produces only a single decision sequence [4]. For the knapsack problem, the standard greedy strategy is *greedy by density*: at each step it selects the object with the largest profit-to-weight ratio p_i/w_i [5]. This is fast and intuitive, and it mirrors how a player picks the strongest pieces that still fit.

However, greedy by density is not guaranteed to be optimal. Consider a knapsack of capacity $M = 4$ with three objects: A with $(w, p) = (3, 5)$, B with $(2, 3)$, and C with $(2, 3)$. The densities are $5/3 \approx 1.67$ for A and 1.5 for B and C, so greedy selects A first; the remaining capacity of 1 then fits neither B nor C, yielding a profit of 5. The true optimum, however, is $B + C$, which fills the capacity exactly for a profit of 6. The greedy choice of the single densest object blocks the better combination. This gap between the greedy result and the true optimum is precisely what this paper measures for the armor-selection problem.

III. Problem Formulation

This section casts the armor-selection task in Elden Ring as an instance of the Multiple-Choice Knapsack Problem.

A character equips armor in four slots: head, chest, arms, and legs. Let $G = \{1, 2, 3, 4\}$ index these slots, and let each slot g contain a set of candidate pieces, where piece i in slot g has a weight $w_{g,i}$ and a value $v_{g,i}$ for the protective attribute the player wishes to maximize, for example, physical damage negation, magic negation, or poise. A binary decision variable $x_{g,i} \in \{0, 1\}$ equals 1 if piece i is equipped in slot g and 0 otherwise. Exactly one piece is chosen in each slot, so each group contributes exactly one piece to the loadout.

The objective is to maximize the total value of the chosen attribute across all slots, subject to the combined weight not exceeding the armor weight budget W:

$$\text{maximize} \quad \sum_{g \in G} \sum_{i} v_{g,i}\, x_{g,i} \tag{5}$$

$$\text{subject to} \quad \sum_{g \in G} \sum_{i} w_{g,i}\, x_{g,i} \le W, \tag{6}$$

$$\sum_{i} x_{g,i} = 1, \quad \forall g \in G, \tag{7}$$

$$x_{g,i} \in \{0, 1\}. \tag{8}$$

The budget W follows from the game's Equip Load mechanic. A character's maximum Equip Load L is determined by the Endurance attribute, and movement depends on the fraction of L that is used: a load up to 30% of L permits a light

(fast) roll, up to 70% a medium roll, and below 100% a heavy roll, while reaching 100% leaves the character unable to roll. Since total Equip Load also includes non-armor equipment such as weapons, the weight available for armor is

$$W = \rho \cdot L - w_{\text{other}}, \tag{9}$$

where ρ is the target load ratio (e.g. 0.70 for a medium roll) and w_{other} is the weight of the equipped non-armor gear. In this study W is treated as an input parameter, so the budget can be varied to examine how the optimal loadout changes.

Solving this model yields the loadout, exactly one piece per slot, that attains the maximum possible value of the chosen attribute within the weight the character can carry.

IV. Methodology

A. Dynamic Programming Solution

The model from Section III is solved exactly with dynamic programming over the four slots, processed as groups $g = 1, \ldots, G$ with $G = 4$. Let $dp[g][c]$ denote the maximum attribute value achievable using slots 1 through g within a remaining weight capacity c, given that exactly one piece is chosen in every slot. The base case considers only the first slot,

$$dp[1][c] = \max_{i \in 1,\, w_{1,i} \le c} v_{1,i}, \tag{10}$$

left undefined (treated as $-\infty$) when no piece in slot 1 fits within c. For $g = 2, \ldots, G$,

$$dp[g][c] = \max_{i \in g,\, w_{g,i} \le c} \big(dp[g-1][c - w_{g,i}] + v_{g,i}\big), \tag{11}$$

where the maximization ranges only over pieces for which $dp[g-1][c - w_{g,i}]$ is defined. Because every slot must be filled, the optimal value is not simply $dp[G][W]$ but the best value reachable at any capacity up to the budget,

$$V_{\text{opt}} = \max_{0 \le c \le W} dp[G][c], \tag{12}$$

since the cheapest feasible loadout need not use the full budget. The chosen loadout is recovered by backtracking from the capacity c^* attaining V_{opt}: for each slot, the piece that produced the stored value is identified, its weight subtracted, and the process continues to the previous slot. The full pseudocode is given in Algorithm 1.

Because armor weights are given to one decimal place (e.g., 4.5), every weight and the budget W are multiplied by ten so that the capacity index c becomes an integer, as the table requires; this only rescales the unit of weight and does not affect the result.

Algorithm 1 Dynamic programming for the armor-loadout MCKP

Require: slots $1, \ldots, G$ with candidate pieces $(w_{g,i}, v_{g,i})$; budget W
Ensure: the optimal value V_{opt} and the loadout attaining it

```
initialize dp[g][c] ← −∞ for all g ∈ {1,...,G}, c ∈ {0,...,W}
for each piece i in slot 1 with w_{1,i} ≤ W do
    dp[1][w_{1,i}] ← max(dp[1][w_{1,i}], v_{1,i})
end for
for g = 2 to G do
    for each c with dp[g−1][c] ≠ −∞ do
        for each piece i in slot g with c + w_{g,i} ≤ W do
            dp[g][c + w_{g,i}] ← max(dp[g][c + w_{g,i}], dp[g−1][c] + v_{g,i})
        end for
    end for
end for
c* ← arg max_{0≤c≤W} dp[G][c]
recover the chosen piece for each slot by backtracking from (G, c*) to slot 1
return dp[G][c*] and the recovered loadout
```

B. Greedy Baseline

To represent the manual, intuitive approach a player would take, the greedy baseline applies the greedy-by-density rule [5] slot by slot, in the fixed order head, chest, arms, legs. For the slot currently being filled, candidate pieces are sorted in decreasing order of the value-to-weight ratio $v_{g,i}/w_{g,i}$ for the chosen attribute. A candidate is accepted if its weight does not exceed the capacity remaining after the slots already filled, and if enough capacity would still remain afterward to cover at least the lightest piece available in every slot yet to be filled; the first candidate in sorted order meeting both conditions is equipped. If no candidate satisfies both conditions, the lightest available piece in that slot is equipped instead, so that every slot is always filled and the procedure never has to backtrack across slots. This guarantees a complete, weight-feasible loadout in a single forward pass, but, like any greedy method, with no guarantee that the loadout is optimal. The procedure is given in Algorithm 2.

C. Top-N Loadout Enumeration

Beyond the single optimal loadout, the implementation also offers the player a ranked list of the N best loadouts, obtained with a branch-and-bound search [5] adapted to the multiple-choice structure. A suffix dynamic-programming table is computed once, processing the slots in reverse order $G, \ldots, 1$, giving for every slot g and every remaining capacity the best attainable value using slots g through G; this table serves as an upper bound for any partial loadout that has already filled slots 1 through $g-1$. Partial loadouts are expanded slot by slot from a priority queue ordered by this upper bound, so the most promising partial loadout is always extended next, and a branch is discarded once its upper bound falls below the value of the weakest loadout currently kept among the top N. The search stops once N complete loadouts have been confirmed and ranked. The first loadout returned coincides with the optimal value V_{opt} from Section IV-A, while the remaining $N-1$ give the player practical, near-optimal alternatives. The procedure is given in Algorithm 3.

Algorithm 2 Greedy-by-density baseline

Require: slots $1, \ldots, G$ in fixed order with candidate pieces $(w_{g,i}, v_{g,i})$; budget W
Ensure: a complete loadout, one piece per slot

$remaining \leftarrow W$
for $g = 1$ to G **do**
  sort candidates in slot g by $v_{g,i}/w_{g,i}$, decreasing
  $minNeeded \leftarrow \sum_{g'>g} \min_i w_{g',i}$
  $picked \leftarrow$ **none**
  for each candidate i in sorted order **do**
    if $w_{g,i} \le remaining$ **and** $remaining - w_{g,i} \ge minNeeded$ **then**
      $picked \leftarrow i$; **break**
    end if
  end for
  if $picked =$ **none then**
    $picked \leftarrow$ lightest candidate in slot g
  end if
  equip $picked$ in slot g; $remaining \leftarrow remaining - w_{g,picked}$
end for
return the equipped loadout

D. Data and Experimental Setup

The experiments use a dataset of 568 armor pieces compiled from publicly available game data [3], grouped by slot into 167 helms, 203 chest pieces, 94 gauntlets, and 104 leg pieces. Missing attribute cells, which occur mainly for "altered" variants, are treated as zero. The featured objective is physical damage negation, although the program accepts any attribute as the optimization target.

The implementation is written in Python, and the dynamic-programming and greedy methods are run on the same inputs. The armor weight budget W is swept from $W = 6$, near the minimum weight at which any complete loadout is feasible, up to $W = 65$, beyond which the optimal value no longer changes, with the finer grid used in Fig. 1 and the breakpoints in Table I reported in detail. For each budget the total attribute value of both the optimal and the greedy loadout is recorded. The optimality gap is computed as

$$\text{gap} = \frac{V_{\text{opt}} - V_{\text{greedy}}}{V_{\text{opt}}} \times 100\%, \tag{13}$$

where V_{opt} and V_{greedy} are the attribute values achieved by the dynamic-programming and greedy loadouts, respectively.

V. Results and Analysis

A. Optimality Gap Across Equip-Load Budgets

Table I and Fig. 1 compare the two methods on the featured attribute, physical damage negation, over the dataset of 568 armor pieces. At very tight budgets ($W \le 7$, not shown in the table) every feasible loadout is forced to use the cheapest piece in each slot, so the DP-optimal and greedy loadouts coincide and the gap is 0%. As soon as the budget loosens slightly, the two methods diverge: at $W = 8$ the gap is already 9.1%, and it grows steadily as W increases, reaching 40.0% at $W = 20$ and 65.1% at $W = 60$, after which the gap stays flat because the DP-optimal value itself stops improving (the program has already assembled the single best-possible loadout for this attribute, 43.0, using 57.7 of the available budget).

The reason for this growing gap is visible directly in Fig. 1: the greedy curve flattens out at a value of 15.0 from

Algorithm 3 Branch-and-bound enumeration of the top N loadouts

Require: slots $1, \ldots, G$ with candidate pieces $(w_{g,i}, v_{g,i})$; budget W; N
Ensure: the N loadouts with the highest total value

build $suffix[g][c] \leftarrow$ best value using slots $g, \ldots, G$ within capacity c, for $g = G, \ldots, 1$
$results \leftarrow \emptyset$; $worst \leftarrow -\infty$
push $(bound = suffix[1][W], value = 0, g = 1, used = 0, loadout = \langle\rangle)$ onto priority queue Q
while Q is not empty **do**
  pop the node with largest $bound$ from Q
  if $|results| \ge N$ **and** $bound < worst$ **then**
    break
  end if
  if $g = G + 1$ **then**
    add $loadout$ to $results$ with its $value$
    if $|results| \ge N$ **then**
      keep only the N best; $worst \leftarrow$ the N-th best value
    end if
    continue
  end if
  for each candidate i in slot g with $used + w_{g,i} \le W$ **do**
    $newValue \leftarrow value + v_{g,i}$
    $newBound \leftarrow newValue + suffix[g+1][W - used - w_{g,i}]$
    if $|results| < N$ **or** $newBound \ge worst$ **then**
      push $(newBound, newValue, g + 1, used + w_{g,i}, loadout \cup \{i\})$ onto Q
    end if
  end for
end while
return the N loadouts in $results$, ranked by value

TABLE I
DP-optimal vs. greedy physical damage negation across armor weight budgets

Budget W	DP defense	Greedy defense	Gap (%)
8	11.0	10.0	9.1
10	14.0	12.0	14.3
15	19.0	15.0	21.1
20	25.0	15.0	40.0
30	33.0	15.0	54.5
40	39.0	15.0	61.5
50	41.0	15.0	63.4
60	43.0	15.0	65.1

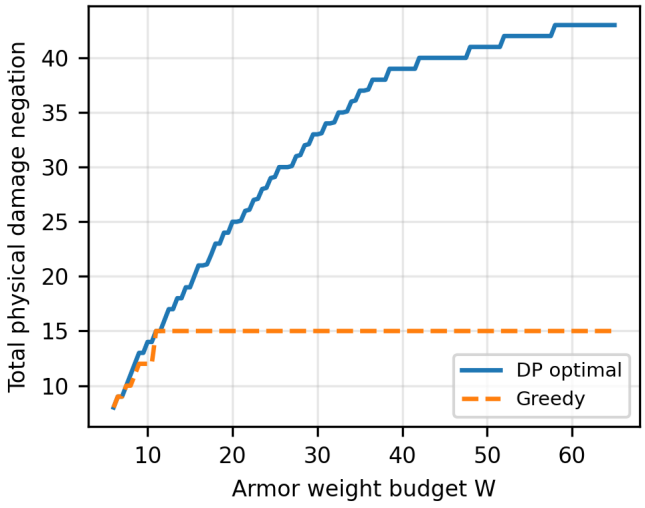

Fig. 1. Maximum achievable physical damage negation as a function of the armor weight budget W, comparing the DP-optimal loadout against the greedy heuristic.

$W = 15$ onward and never rises again, while the DP curve keeps climbing almost the entire way to $W = 60$. Greedy commits, independently in each slot, to the single piece with the highest value/weight ratio that is feasible; once that piece is affordable, it remains the greedy choice no matter how much additional capacity becomes available, because greedy never reconsiders a slot once it has been filled. Dynamic programming, in contrast, is free to trade a high-density piece for a heavier, higher-absolute-value piece whenever the extra weight is affordable, which is exactly what drives its curve upward as W grows. The largest measured gap in this experiment is 65.1%, at $W = 57.7$ and beyond—meaning that, in the worst case observed, a player who manually picks pieces by their value-to-weight ratio ends up with barely a third of the physical damage negation that is actually achievable within the same equip-load budget.

B. Optimality Gap Across Target Attributes

The previous experiment fixes the target attribute to physical damage negation. Table II repeats the same comparison at a fixed mid-range budget, $W = 30$, for every attribute the program can optimize, to check whether a 65%-scale gap is specific to that one attribute or a general property of the model.

The gap is not a fixed property of the model; it ranges from 5.7% (Immunity, Poise) to 54.5% (Phy) at the same budget. The "Greedy w" column, the weight the greedy loadout actually uses out of the $W = 30$ available, explains most of this spread. For Poise, greedy already spends the entire budget (30.0 of 30): the highest-density piece in each slot happens to also be expensive enough that no weight is left unused, so there is no room for dynamic programming to do better and the gap collapses to the small amount lost from greedy's per-slot, rather than global, choice. For Phy, Robustness, and Pierce, greedy instead leaves between 18 and 21 units of budget completely unused, because the highest-density pieces for these attributes happen to be light; dynamic programming spends almost all of that unused weight on heavier, higher-value pieces, which is what drives the gap above 50%. Immunity is the interesting middle case: greedy also leaves substantial slack (18.6 unused), but the DP-optimal loadout for Immunity already saturates at a weight of only 19.0, there simply is not much more Immunity to be gained from spending more weight, so the gap stays small even though greedy is clearly not using the full budget. The general pattern is that the size of the optimality gap depends jointly on how much weight slack the greedy heuristic leaves unused and on how much additional value is actually available within that slack, not on the attribute's nominal scale.

TABLE II
DP-OPTIMAL VS. GREEDY VALUE BY TARGET ATTRIBUTE AT $W = 30$

Attribute	DP value	Greedy value	Greedy w	Gap (%)
Phy	33.0	15.0	10.7	54.5
Robustness	186.0	91.0	8.8	51.1
Pierce	34.0	17.0	11.4	50.0
Strike	30.3	15.5	9.0	48.8
Slash	37.1	22.0	13.6	40.7
Focus	244.0	171.0	7.2	29.9
Fire	33.0	24.0	6.2	27.3
Magic	32.0	25.0	6.0	21.9
Holy	31.1	25.0	5.9	19.6
Light	29.0	24.0	6.2	17.2
Vitality	249.0	217.0	6.7	12.9
Immunity	245.0	231.0	11.4	5.7
Poise	53.0	50.0	30.0	5.7

TABLE III
AVERAGE RUNNING TIME ON THE FULL DATASET

Budget W	DP (ms)	Greedy (ms)	B&B top-5 (ms)
10	2.09	0.07	1.73
30	8.03	0.07	6.93
60	11.54	0.07	17.54

C. Computational Cost

Table III reports wall-clock running time on the full 568-piece dataset, averaged over repeated runs on the same machine, for the physical-damage-negation objective at three representative budgets.

The measurements agree with the complexity discussed in Section II-C: dynamic programming runs in $O(G \cdot W \cdot T)$ time, where T is the largest number of candidates in a slot, so its running time grows roughly linearly with the integer-scaled budget W, as seen going from $10\,\text{ms}$ at $W = 10$ to $60\,\text{ms}$-scale at $W = 60$ (in the integer-scaled units used internally, $W = 60$ corresponds to a capacity index of 600). Greedy stays under a tenth of a millisecond at every budget tested, consistent with its $O(G \cdot T \log T)$ cost, which does not depend on W at all. The branch-and-bound search for the top-5 loadouts costs more than plain DP once W is large, because a larger budget admits more partial loadouts whose upper bound is competitive enough that the priority queue cannot prune them

TABLE IV
LOADOUT CHOSEN BY EACH METHOD, PHY, $W = 20$

Slot	Piece	Weight	Phy
DP-optimal (total Phy $= 25.0$*, weight* $= 19.9$*)*			
Helm	Blue Silver Mail Hood	3.8	4.0
Chest Armor	Vagabond Knight Armor (altered)	9.6	13.0
Gauntlets	Gold Bracelets	0.8	1.0
Leg Armor	Exile Greaves	5.7	7.0
Greedy (total Phy $= 15.0$*, weight* $= 10.7$*)*			
Helm	Blue Silver Mail Hood	3.8	4.0
Chest Armor	Aristocrat Garb (altered)	4.1	7.0
Gauntlets	Gold Bracelets	0.8	1.0
Leg Armor	Cloth Trousers	2.0	3.0

immediately; even so, the slowest measurement is still under $20\,\mathrm{ms}$, well within the latency a player would expect from an interactive tool. All three methods comfortably scale to the full, unfiltered armor dataset, so restricting the search to a player's owned items, as the implementation also supports, only makes every method faster.

D. Illustrative Case Study

To make the abstract gap numbers concrete, Table IV lists the actual loadout chosen by each method for physical damage negation at $W = 20$, where Table I reports a 40% gap.

Both methods agree on the Helm and Gauntlets, where the highest-density piece is also the best available choice. The difference is entirely in the Chest Armor and Leg Armor slots: greedy keeps the lightest, highest-ratio pieces in both (4.1 and 2.0 weight), leaving 9.3 of the 20-unit budget unused, while dynamic programming spends that unused weight on a heavier chest piece (9.6 weight, 13.0 Phy instead of 7.0) and a heavier leg piece (5.7 weight, 7.0 Phy instead of 3.0), for a net gain of 10.0 Phy at the cost of 9.2 additional weight—weight that greedy had simply left on the table.

This budget also illustrates why the top-N feature from Section IV-C is useful beyond just reporting a single number. At $W = 20$, five distinct loadouts tie for the optimal value of 25.0: they share the same Chest Armor and Gauntlets but differ in the Helm (Blue Silver Mail Hood or Chain Coif) and the Leg Armor (Exile Greaves, Malenia's Greaves, or Vagabond Knight Greaves), all of equal weight and Phy contribution. A player using only the single DP-optimal answer would see one specific combination; the branch-and-bound search surfaces all five, which is valuable in practice because the pieces differ in their other stats (e.g., resistances or poise) even though they are equivalent for physical damage negation, letting the player break the tie according to a secondary preference the model does not explicitly optimize.

VI. CONCLUSION

This paper formalized armor selection in Elden Ring as a Multiple-Choice Knapsack Problem, with one group per equipment slot and exactly one piece chosen from each, and solved it exactly with dynamic programming in pseudo-polynomial time. Compared against the greedy-by-density rule that mirrors how players choose equipment by hand, the dynamic-programming solution is never worse and, on the physical-damage-negation objective over a dataset of 568 armor pieces, outperforms greedy by as much as 65.1% once the equip-load budget is generous enough for the two methods to diverge. Repeating the comparison across every attribute the program supports shows that the gap is not specific to one statistic: it ranges from a negligible 5.7%, when the greedy loadout already happens to spend the entire budget, up to more than 50% for several damage-negation types, when greedy leaves a large part of the budget unused. The underlying reason is structural rather than incidental: greedy commits to the single highest-density piece in each slot and never revisits that choice as more capacity becomes available, whereas dynamic programming keeps exploring higher-value, heavier alternatives for as long as the budget allows, and it does so in tens of milliseconds even on the full, unfiltered dataset, so the exact method is not just more accurate but also fast enough for interactive use. This confirms that the manual, intuition-based loadout selection many players rely on can leave a substantial amount of the chosen attribute on the table, and that an exact algorithm closes that gap without added effort from the player.

The current model optimizes a single attribute at a time and treats the armor weight budget as already separated from the rest of the character's equipment. Future work could extend the formulation to a multi-objective setting that balances poise against one or more damage-negation values simultaneously, for instance through a weighted-sum or Pareto-front extension of the same dynamic-programming recurrence, and could fold weapons and talismans into the same weight budget so that the optimization covers the player's entire loadout rather than armor alone.

ACKNOWLEDGMENT

The author wishes to express his sincere gratitude to Prof. Dr. Ir. Rinaldi, M.T., whose insightful lectures in the IF2211 Strategi Algoritma course at Institut Teknologi Bandung laid the conceptual foundation on which this work is built. The author is also grateful to family, friends, and peers for their continued encouragement and support throughout the writing of this paper.

APPENDIX

The full source code of the armor-loadout optimizer described in this paper, together with the armor dataset and instructions for running it, is publicly available at:

https://github.com/Nusss0/EldenRing-Armor-Optimizer

REFERENCES

[1] Anime News Network, "FromSoftware's Elden Ring Game Sells 30 Million Copies," Apr. 28, 2025. [Online]. Available: https://www.animenewsnetwork.com/news/2025-04-28/fromsoftware-elden-ring-game-sells-30-million-copies/.224012

[2] Statista, "Lifetime unit sales of Elden Ring: Shadow of the Erdtree worldwide as of July 2025," 2025. [Online]. Available: https://www.statista.com/statistics/1620096/elden-ring-shadow-of-the-erdtree-sales-worldwide/

[3] Robikscube, “Elden Ring Ultimate Dataset,” Kaggle, 2022. [Online]. Available: https://www.kaggle.com/datasets/robikscube/elden-ring-ultimate-dataset
[4] R. Munir, “Program Dinamis (Dynamic Programming),” Bahan Kuliah IF2211 Strategi Algoritma, Program Studi Teknik Informatika, Institut Teknologi Bandung, 2026.
[5] R. Munir, “Algoritma Branch and Bound,” Bahan Kuliah IF2211 Strategi Algoritma, Program Studi Teknik Informatika, Institut Teknologi Bandung, 2026.
[6] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems*. Berlin, Germany: Springer, 2004.

STATEMENT OF ORIGINALITY

I hereby declare that this paper is my own writing, not an adaptation, translation, or plagiarism of another person’s work.

Bandung, 18 Juni 2026
Stevanus Agustaf Wongso – 13524020