Application of Dynamic Programming with Knapsack-based Approach in Optimizing Gun Economy on Valorant

Nathan Edward Christofer Marpaung - 13524062
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: 011nathanmarpaung@gmail.com , 13524062@std.stei.itb.ac.id

***Abstract*— Valorant is a tactical first-person shooter game that incorporates a complex economic system in which players must manage credits to purchase weapons, armor, and abilities before each round. Proper credit management and allocation is essential for maintaining competitive advantage while preserving long-term economic stability throughout a match. This paper proposes an optimization framework for Valorant gun economy using Dynamic Programming and a Knapsack-based approach. The weapon purchasing process is modeled as a Multiple-Choice Knapsack Problem (MCKP), where available credits represent capacity, equipment costs represent weights, and combat effectiveness scores represent item values. Combat effectiveness is evaluated through a utility function that incorporates weapon statistics, agent–weapon synergies, map characteristics, team side advantages, enemy weapon compositions, player performance metrics, and round-specific economic strategies. Dynamic Programming is then applied to determine the optimal combination of primary weapons, sidearms, and armor under budget constraints while considering save targets as soft constraints. Experimental test cases demonstrate that the proposed approach is capable of generating effective loadout recommendations that maximize combat utility without exceeding available credits. The results indicate that the model can successfully balance immediate combat effectiveness and long-term economic preservation. Although the system relies on heuristic scoring and static game data, the proposed framework demonstrates the applicability of Dynamic Programming and Knapsack optimization techniques to strategic decision-making problems in competitive games.**

Keywords*— *Dynamic Programming, Multiple-Choice Knapsack Problem, Valorant, Gun Economy Optimization, Resource Allocation, Game Strategy.

I. Introduction

Valorant is one of the most popular tactical First-Person Shooter (FPS) games in the world. In Valorant, player aims to kill and defeat an enemy using precise aim and guns until a certain set amount of rounds. In order to give players certain edges and advantages, Valorant features a complex economic system where players are able to earn credits through kills and rounds and spend these credits to purchase weapons, shields, and abilities before each round. A proper balance between player skill and efficient credit management allows players to have significant advantages in winning rounds and games in Valorant.

Fig 1.1 Valorant (Source : Riot Games)

The gun economy in Valorant relies on optimizing credit budget by selecting the best combination of weapon, abilities, and shield within the budget. Overusing credits can result in lower-tier weapons being used every round, while oversaving credits can result in losing rounds and momentum to enemy team. By optimizing spending strategies to maximize combat efficiency while preserving economic stability, players are able highest level of effectiveness and higher chance of winning rounds while securing better weapons more.

This paper aims to applicate Dynamic Programming with Knapsack-based approaches to optimize credit allocation towards weapons, shields, and abilities in order to provide the best decision for players and providing the best case scenario to win rounds and preserve gun economy.

II. THEORETICAL BASIS

A. Valorant Gun Economy

Valorant is a tactical FPS that incorporates an economy system for players to buy guns and abilities. Players have the option to buy a variety of guns that have their own price, strength, and quirks. Players are also able to buy special abilities owned by the character that the players choose to play as, and shields as a measure to protect themselves from incoming damage. All of these abilities and guns are bought by players

using the match credit budget each player earned at the start of the round.

Fig 1.2 Valorant Buy Menu (Source : Author)

In Valorant, match credits are earned round-by-round by players based on their individual and team performance in the previous round. At the start of a match, each player within each team are provided 1000 credits on the starting round to be used to buy guns and abilities. Afterwards, players are rewarded credits based on their team performance and individual performance at the start of the next round.

Each round wins will provide 3000 credits for the winning team, while the losing team are rewarded 1900 credits that gradually increase by +500 credits up to 2900 credits. If a player manage to secure kills, they will be rewarded 200 credits per kill. If the attacker team managed to secure the objective and plant the spike, each player on the attacker team wil be rewarded 300 credits. If one of the team failed to achieve their objective (failure to plant the spike for the attacker team, or failure to defuse the spike the defender team) while surviving the round, they will get a penalty and be provided only 1000 round.

Each weapon will have different strength and characteristics, such as different fire rate, damage, or certain quirks like a sniper or an assault rifle. The difference between combat capabilities result in importance of managing economy and credits to ensure players have the best combat capabilities and chance to win duels against enemies, or preserve credits to ensure better weapons the next few rounds. Managing gun economy becomes a very integral strategy within Valorant in ensuring winning rounds overtime or shifting momentum from losing into winning.

B. Optimizing Gun Economy in Valorant

An optimization problem aims to determine the best solution among a set of feasible options that satisfy a set of specified constraints. Generally, optimization involves in maximizing and/or minimizing an objective function or object in accordance to the specified constraints.

In the context of Valorant, optimizing gun economy means maximizing the effectiveness of the combination of weapons and abilities bought based on the buying strategy chosen by the player and/or team. The objective is to provide the best decision in weapon purchasing without exceeding the available credits. Consequently, weapon purchasing can be formulated as a constraint optimization problem

Different buying strategies results in different optimizations and decision-making. An “eco round” means saving as much money as possible to ensure better weapons for the next round. A “force buy” round means using the majority of your minimal credits to buy medium-tier weapons to try forcing a win in the current round. A “half-buy round” means using your credits to buy medium-tier weapon while still saving a handful of match credits to be used in the next round. A “full buy round” means using your credits to buy the best weapons, abilities, and armor and maximize your combat capabilities.

C. Dynamic Programming

Dynamic Programming (DP) is an algorithmic technique that mainly does optimization over plain recursion. The idea is to decompose problems into common smaller sub-problems that overlaps and are able to be combined into a single optimized solution. Each sub-problem are solved only once and the results are stored into a table to be reused when needed. Each solve will gradually build an optimized solution based on the objective and constraint provided for the problem[5].

Dynamic Programming are mainly used for problems with the following characteristics:

1. Optimal Substructure: Using the optimal results of subproblems to achieve the optimal result of the bigger problem. [1,5]
2. Overlapping Subproblems: Each subproblems are solved repeatedly in different parts of other subproblems such that each values are being used repeatedly. [1,5]

Dynamic Programming focuses on eliminating the need of recalculating reoccurring subproblems and values while also breaking down complex, hard problems into smaller problems that are much easier to solve, thus saving computing power and time. To achieve that, there are 2 approaches in implementing dynamic programming:

1. Top-Down Approach (Memoization): Problems are solved recursively with each solved value are stored in memorization table where each problem will check the table first for any reoccurring value before calling any recursive calls[1].
2. Bottom-Up Approach (Tabulation): Problems are solved starting from the smallest subproblems that gradually build up to the final solution[1]. A table stores the solution for the base case of the subproblems. An iterative solution are made to fill the remaining entries on the table using the base case values stored in the table.

The principles of Dynamic Programming points to building solutions recursively and incrementally by using stored values inside tables. The concept of Dynamic Programming can be applied into different set of problems to help simplify problems and find the optimal solution for the problem, such as Knapsack problems.

D. Knapsack Problem

The Knapsack Problem is a well-known problem where Dynamic Programming can be applied in finding the optimal solution[3]. The problem consists of n distinct items with varying values and weight, and a knapsack of capacity W [2]. Solving the problem consists of finding a subset combination of items where the total weight of the item doesn't exceed capacity W. An optimal solution for the Knapsack Problem is a subset of item where the total value of item picked is the best compared to subsets without exceeding capacity W.

Assume there is n items and a knapsack with capacity of W. Each item can be modeled as items with value and weight:

$$item_i = (v_i, w_i)$$

The optimization objective can be defined as:

$$Maximize\ Z = \sum_{i=1}^{n} \mathrm{v_i} x_i$$

Where

$$\sum_{i=1}^{n} w_i x_i \leq W$$

There are different variants of the knapsack problem:

1. 0/1 Knapsack Problem : Most common variant where there are distinct N items where each item can only be picked once and as a whole[2,3].
2. Fractional Knapsack Problem : There are distinct N items and each item can only be picked once, but items are allowed to be picked up fractionally (not as a whole)[3].
3. Unbounded Knapsack Problem : Similar to 0/1 Knapsack Problem but we are able to pick up the same item without any bounds[3].

This paper will focus on using 0/1 Knapsack Problem as a methodology in approaching Valorant's gun economy problem with extra conditionals and constraint applied in deciding equipment to buy.

E. Modelling Valorant Gun Economy to Knapsack Problem

Valorant Gun Economy can be modelled into a Knapsack Problem, focusing on the weapon purchasing process that happens within the start of a round in Valorant. Game elements within the ability and weapon purchasing process can be corresponded to knapsack components:

TABLE I. GAME ELEMENTS MAPPED TO KNAPSACK

Knapsack Components	Valorant Game Elements
Item	Weapon/Abilities
Weight	Credit Cost
Value	Combat Capability Score
Capacity	Available Credits

By formulating valorant game elements into this manner, an optimal solution towards decision making in buying weapons and abilities can be found using Dynamic Programming with Knapsack-based approach. The optimal solution will be the best combination of weapon and abilities to buy based on the strategy picked by the player (force buy round, eco round, half-buy round, or full buy round). There will be extra constraints and conditionals set to ensure the most optimal solution based on the gamestate of the player, such as only 1 primary weapon and weapon-character synergies.

III. METHODOLOGY AND IMPLEMENTATION

A. Data Procurement

Knapsack Problems are reliant with deciding the value of of each items. Dynamic Programming helps us in storing these calculated value to reduce needed computational power. In this papers context, the utility function determined for Valorant will be the combat capabilities of the combination of weapons and abilities/utilities bought by the player.

Before implementing utility calculation, each weapon and abilities must be determined first. Each weapon in Valorant have their own credit cost, capabilities and unique quirks, different damage, and different playstyles. Each of these capabilities and unique quirks must be considered inside the utility function, so combat data and statistics must be stored inside the program.

Combat statistics can be gathered from Valorant patch notes and wiki websites. This paper will gather data from Valorant patch note 12.11, and statistics from each weapon are gathered from wiki websites such as mobalytics.gg and valking.gg.

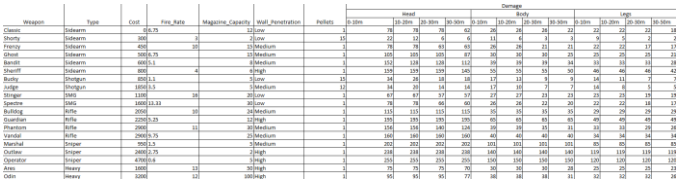
Weapons are divided into 4 types of weapon: Sidearm, SMG, Rifle, Sniper. Rifles focus on precision shooting and consistent damage, SMG are focused on fast fire rate and close range duels, Sniper focuses on long range damage and shooting. All three types of those weapon are primary weapons, while Sidearm are secondary weapons. Each player can carry one primary weapon and one secondary weapon on every round.

As previously mentioned, each weapon will have their own cost and combat capabilities. Most notably, each weapon will have different fire rate, magazine size, damage based on range, number of pellets, and degree of wall penetration. These statistics will primarily dictate the combat strength a player has, and utility + weapon synergies will dictate the winning chance of a player in a duel. With that in mind, these statistics are gathered and stored into a singular table.

TABLE II. VALORANT WEAPON DATA EXAMPLE

Weapon	Type	Cost	Fire Rate	…
Classic	Sidearm	0	6.75	…
Vandal	Rifle	2900	9.75	…

With data from Valorant wiki websites on the 12.11 patch note, the resulting weapon data are:



Weapon	Type	Cost	Fire_Rate	Magazine_Capacity	Wall_Penetration	Pellets	Damage											
							Head				Body				Legs			
							0-10m	10-20m	20-30m	30-50m	0-10m	10-20m	20-30m	30-50m	0-10m	10-20m	20-30m	30-50m
Classic	Sidearm		6.75	12	Low	1	78	78	78	62	26	26	26	22	22	22	22	18
Shorty	Sidearm	300		2	Low	1	22	12	6	6	11	6	3	3	9	5	2	2
Frenzy	Sidearm	450		10	Medium	1	78	78	69	69	26	26	23	23	22	22	17	17
Ghost	Sidearm	500	6.75	15	Medium	1	105	105	105	87	30	30	30	25	25	25	25	21
Bandit	Sidearm	600	5.1	8	Medium	1	152	128	128	112	39	39	39	34	33	33	33	28
Sheriff	Sidearm	800		6	High	1	159	159	159	145	55	55	55	50	46	46	46	42
Bucky	Shotgun	850	1.1	5	Low	15	34	34	34	18	17	13	9	9	14	11	7	7
Judge	Shotgun	1850	3.5	5	Medium	12	34	20	14	14	17	10	7	7	14	8	5	5
Stinger	SMG	1100	16	20	Low	1	67	67	57	57	27	27	23	23	25	25	19	19
Spectre	SMG	1600	13.33	30	Low	1	78	78	66	60	26	26	22	20	22	22	18	17
Bulldog	Rifle	2050		24	Medium	1	115	115	115	115	33	33	33	33	29	29	29	29
Guardian	Rifle	2250	5.25	12	High	1	195	195	195	195	65	65	65	65	49	49	49	49
Phantom	Rifle	2900		30	Medium	1	156	156	140	124	39	39	35	31	33	33	29	28
Vandal	Rifle	2900	9.75	25	Medium	1	160	160	160	160	40	40	40	40	34	34	34	34
Marshal	Sniper	950	1.5	5	Medium	1	202	202	202	202	101	101	101	101	85	85	85	85
Outlaw	Sniper	2400	2.75	2	High	1	238	238	238	238	140	140	140	140	119	119	119	119
Operator	Sniper	4700	0.6	5	High	1	255	255	255	255	150	150	150	150	120	120	120	120
Ares	Heavy	1600		50	High	1	75	75	75	70	30	30	30	28	25	25	25	23
Odin	Heavy	3200		100	High	1	95	95	95	77	38	38	38	31	32	32	32	26

Fig 3.1 Valorant Weapon Data (Source: Author's document, data procured from mobalytics.gg)

Since the data are still stored in an excel file, the data are converted into a JSON format for easier usage and easier import to the main program.

```
...
 {
    "weapon": "Operator",
    "type": "Sniper",
    "cost": 4700,
    "fire_rate": 0.6,
    "magazine_capacity": 5,
    "wall_penetration": "High",
    "pellets": 1,
    "damage": {
      "head": {
        "0-10m": 255,
        "10-20m": 255,
        "20-30m": 255,
        "30-50m": 255
      },
      "body": {
        "0-10m": 150,
        "10-20m": 150,
        "20-30m": 150,
        "30-50m": 150
      },
      "legs": {
        "0-10m": 120,
        "10-20m": 120,
        "20-30m": 120,
        "30-50m": 120
      }
    }
  },
'''
```

Converted JSON file

With the data fully converted into a JSON format, the data are able to be imported to the main program easily and stored inside data structures for the program to manage. The main program will read the JSON file and store the data values inside a class struct called Weapon. Each of these data structs are then stored inside a list. An enum struct are also added to identify weapon types. A dictionary is also added to search and get weapon data using weapon names as key.

B. Constraints in Evaluating Combat Score

Game contexts are an important aspect in determining buying and weapon strategies. Maps, agents, and team sides largely influence gameplan and strategies, which include weapon usage and playstyles. This means game context such as maps and agents are detrimental towards deciding combat and utility value and should be equated in the utility function.

Agent used by players also have varying abilities that provide different strengths and capabilities. These capabilities can provide unique playstyles and extra synergies with weapons. An agent could have better mobility that provide them better chances on winning duels up close or a dash to take more aggressive gun duels, while other characters might have abilities that help them to take more long range angles and help them escape in an instant. This means certain agents and roles can have abilities that can be considered when picking optimal gun combination as they have better synergies with certain weapons.

In order to equate this synergy into evaluating utility and combat capability, data structs are created to store character names and their abilities. A class struct called AgentMeta holds information on certain capabilities a character might have that can provide better synergy on some weapons, such as mobility abiltiies, info abilities, or flash abilities. This class struct are paired with an enum struct to make a dictionary table where information on meta abilities on an agent by using the agent's name as key.

This dictionary table will then be used in a function to equate synergies between certain types of weapons. The function will check the agent's meta abilities and add multiplier to certain types of weapons that benefit from their abilities. The multiplier will add better heuristic combat value to help in deciding the optimal gun combination and economy.

Maps can also provided different synergies towards certain weapons and strategies. A longer, larger map might benefit snipers a lot more as sniper benefit from long range gun fights, while a smaller map might benefit higher rate weapons such as SMGs as high fire rate weapons benefit the most in shorter range duels. This means map and weapon synergies should be taken into equation when deciding overall combat capabilities of a weapon.

The synergy between maps and weapons can be achieved by adding a multiplier modifier for certain maps. A dictionary struct are added to create a modifier table that the main program can extract and add for certain weapons. The program can access these table by using an enum struct of map names as key values for the dictionary.

Team sides also influence gun play and strategies greatly. An SMG or Rifle could benefit attacking teams more as the attacking team want to initiate more aggressive duels, while snipers could benefit defending teams more as they provide cover and precise damaging weapon from a longer range. This means team sides can provide modifier towards different types of weapons.

The synergy between team sides and weapons can be achieved by making a modifier table for 2 different sides. A dictionary table are made to store different multiplier/modifier values for all 4 types of weapons. The dictionary and multiplier values can then be accessed by the main program by using side names (attacker or defender).

Weapon types can also counter different weapon types. Longer range weapons like snipers counter and wins against shorter range weapons such as SMGs or Shotguns, while SMGs could win better against Rifles in close ranges. This means buying strategies can be influenced by the types of weapons that the enemy team are currently using. This counter-synergy can be achieved by making a dictionary table that will store these modifier values. The table will store modifier values for other types of weapons for each type of weapons. The main program will then access these modifier values based on the type of weapons the enemy are using in order to provide more accurate optimization.

Lastly, weapon choice is greatly influenced by the buying strategy the team or player chose to do at a round. A pistol round means no other types of weapons can be bought except pistols/sidearms. An eco round prefers greater credit savings which means sidearms/pistols will be much more preferred as they are much cheaper than other types of weapons, while a full buy round will prefer the strongest weapons which means rifles and snipers are much more preferred than pistols or SMGs. This means buying strategy must also be taken account when estimating the utility and combat capability value for each gun.

The synergy between buying strategies and weapon types can be achieved by creating a dictionary table that functions as a modifier table that will store different modifier values for each type of weapons. The table stores all 4 multiplier values on each type of weapons for all types of buying strategy (full buy, eco round, pistol round, half-buy round). The main program can then access these modifier values by using RoundType as key value and provide more accurate assessment on gun economy optimization.

There are also different small synergies taken account when it comes to evaluating gun capabilities, such as current player performance (accuracy and headshot rate on the ongoing game), the aggression of the player, or whether the guns and armor the enemy has. These small game contexts can pile up and influence the game and buying strategy together.

To take these smaller contexts into equation, a PlayerState class struct are made to store the current performance and aggression of the player. A TeamState class struct are made to store all of the player states and provide better context in optimizing gun economy such as dropping weapons for another player. An EnemyState class struct are also made to recount the performance and capabilities of the enemy team in taking and winning duels against the player's team. All of these states are then stored inside GameState and GameContext to be evaluated as a whole. These evaluation will provide context on the momentum and strength of each team, thus providing better accuracy in optimizing gun economy.

C. Evaluating Combat Score

Each candidate weapon combat scores are evaluated based on the previous constraints through a multi-component utility function compute_utility. The function takes the weapon and game context as a parameter, which then maps the weapon and the context to a scalar utility value that represents the combat effectiveness for that round. The function will compute a base score from different components and constraints and is organized into two stages.

The first stage involves constructing an additive base score based on alignment between player performance, weapon effective DPS, and map range fit. Weapon effective DPS are derived from the weapon's base damage stats and are scaled with player performance and aggression level. Players with higher performance such as headshot percentage will benefit weapons that provide better damage when hitting a headshot. Map range fits are also applied to penalize weapons that doesn't fit the map and rewards types of weapons that fit the map best.

The second stage involves adding multipliers and modifiers based on the constraints and synergy mentioned previously. The function will look at the current game context and add modifiers towards weapons that best support the current gun economy, such as agent-weapon synergies, enemy weapon counters, and enemy economic pressure. Game states will provide better context on overall weapon value and provide better accuracy on weapon decision making.

The final utility score is the product of the base score alongside the different multipliers, producing a single comparable value for each weapon that represents overall combat effectiveness which maps strategic context, agent identity, team composition, and economic constraints altogether. The final score are then provided and used using dynamic programming to provide weapon selection and decision making that provide the optimal selection to preserve gun economy.

```
    compute_utility(weapon w, context ctx):

        getPlayerStats(ctx)    //    accuracy,    hs_rate,
aggression, agent

        // Stage 1
        getEffectiveDPS(w, ctx) // DPS × accuracy × armor_pen
        getHeadshotBonus(w, ctx) // base_damage × hs_rate if
can_oneshot
        getMapRangeFit(w, ctx) // penalize weapon range vs
map preference
        getPlaystyleFit(w,  ctx)  //  aggression  vs  weapon
fire_rate / range
        base ← eff_dps × range_fit + hs_bonus + style_score

        // Stage 2
        getAbilityWeaponSynergy(w,  ctx)  //  agent  ability
flags × weapon type rules
        getCrossAgentCombo(w, ctx) // teammate pair synergy
bonus
        getRoleCoverageGap(w, ctx) // team role gap bonus
        getEnemyCounter(w, ctx) // enemy playstyle fractions
+ COUNTER_TABLE
        getEcoPressure(w,  ctx)  //  weapon  cost  gradient  vs
enemy credits
        getGameStateMult(w, ctx) // momentum × performance ×
round criticality
        getRoundTypeMult(w, ctx) // filter by pistol / eco /
half-buy / full-buy
        getSideModifier(w,  ctx)  //  attacker  or  defender
weapon bonus
        getSavePenalty(w,  ctx)  //  penalty  if  cost  exceeds
save target

        utility  ←  base  ×  ability_mult  ×  combo_mult  ×
role_mult
                        ×   counter_mult   ×   eco_mult   ×
game_mult
                        × round_mult × side_mult × save_mult
        return utility
```

Pseudocode for computing utility scores

D. Dynamic Programming for Weapon Selection

Dynamic Programming are then used to take all of the computed final utility scores for each weapon to select the best primary weapon to be bought. The pool is first filtered by round type eligibility and budget affordability. Each weapon will represent an item with credit cost as their weight and their utility/combat score as their item value.

There are several constraints or rules regarding weapon selection. First, the game doesn't allow players to bring more than one primary weapon, which means there are at maximum

one primary weapon bought by the player. This also applies towards sidearms and armors. Second, each round will establish forced rules so that weapon selection prioritize buying exactly one option for each category (primary weapon, sidearm, armor) unless it benefits much more to not from one of the categories. With these constraints, we can apply Multiple Choice Knapsack Problem, an extension of 0/1 Knapsack Problem to implement weapon selection in our program. Multiple Choice Knapsack Problem is a generalization of the ordinary knapsack problem, where the set of items is partitioned into different classes [5].

Since there are 3 different loadout decisions, the loadout decision space is partitioned into three groups :

$$G = \{G_0, G_1, G_2\}$$

where G_0 is the primary weapon pool, G_1 is the armor tier pool, and G_2 is the sidearm weapon pool. Each item in each pool will have credit cost as their weight value and their calculated utility as a heuristic value computed beforehand. The objective of MCKP will be to maximize each heuristic combat value for every decision group, where each group are subjected to only 1 option which includes null (no item).

A two-dimensional DP table is then defined as:

$$dp[i][b]$$

Where i denotes the decision group and b denotes each budget slot. A recurrence relation can then be set to fill the DP table, where for each group G_i and each budget slot b,

$$dp[i][b] = \max(dp[i-1][b], \max\{dp[i-1][b - w_j] + u_j\}$$

Where

$$j \in G_{\{i-1\}} \; and \; w_j \leq b$$

The first argument $dp[i-1][b]$ represents the skip decision from the previous group and the state inherits the same budget from the previous group. The second argument iterates all over feasible items j in the current group where its cost does not exceed the current budget and combined with the best achievable utility from prior groups at the reduced budget $b - w_j$ with the current item's utility u_j

In games, players often plan their budgeting for a future round by setting up a minimum credit as save target. As such, the program will take save target as a soft constraint. Item choices can exceed the save target and save less money overall, but exceeding save target will give a penalty towards the utility score of the weapon selected. This means weapon selection will still prioritize saving as much credits while also allowing more weapon selection with less saved credits if the combined utility score surpasses the penalty.

After the DP table are formed through reoccurrence, the solution is reconstructed with a backtracking manner through reverse group order starting at $b = W$.

IV. RESULTS AND ANALYSIS

A. *Results*

Several test cases are made to simulate different player situations and game context. Player States and Game Contexts are inputted and run through the program to be analyzed and optimized regarding buying choice and credit optimization. The resulting choice are then printed into through CLI.

For a player playing as agent Jett doing a full buy round with high amount of credits, the resulting optimization are:

```
VALORANT LOADOUT OPTIMIZER  v6

Agent  : JETT            Map    : BREEZE
Sisi   : DEFENDER        Round  : FULL_BUY
Budget : 5,900 cr        Save target: 0 cr
HS Rate: 35%             Aggression : 25%
Role   : DUELIST         Mobility:True  Info:False  Hold:False


[PRIMARY]  Operator  (SNIPER)
  Biaya    : 4,700 cr
  Utilitas : 792.3
  Base     : 419.2  DPS:300.0  HS:89.2  RangeFit:0.97
  Multiplier breakdown:
    ability_mult  : ×1.4000  (ability rules bonus: +0.4000)
    combo_mult    : ×1.0000  (cross-agent bonus: +0.0000)
    role_mult     : ×1.0800  (role coverage gap)
    counter_mult  : ×1.0000  (vs dominant enemy: unknown)
    eco_mult      : ×1.00    (enemy eco pressure: 0.50)
    game_mult     : ×1.0000  (momentum:0.50 crit:0.50)
    round_mult    : ×1.00    side_mult:×1.25    save_mult:×1.000

[ARMOR]    Heavy Shield  (+50 HP)  1,000 cr

[SIDEARM]  Classic  (sniper backup)  0 cr


Pengeluaran : 5,700 cr  |  Simpan: 200 cr (3%)
Utilitas    : 852.1
```

Fig 4.1 Test Case 1 Results (Source: Author's program)

For a player playing as agent Sova doing a half-buy round with high aggression and high performance, the resulting optimization are:

```
VALORANT LOADOUT OPTIMIZER  v6

Agent  : SOVA            Map    : ASCENT
Sisi   : ATTACKER        Round  : HALF_BUY
Budget : 3,200 cr        Save target: 800 cr
HS Rate: 22%             Aggression : 45%
Role   : INITIATOR       Mobility:False  Info:True  Hold:False


[PRIMARY]  Guardian  (RIFLE)
  Biaya    : 2,250 cr
  Utilitas : 353.5
  Base     : 344.5  DPS:275.7  HS:57.2  RangeFit:0.89
  Multiplier breakdown:
    ability_mult  : ×1.1000  (ability rules bonus: +0.1000)
    combo_mult    : ×1.0000  (cross-agent bonus: +0.0000)
    role_mult     : ×1.0600  (role coverage gap)
    counter_mult  : ×1.0000  (vs dominant enemy: unknown)
    eco_mult      : ×1.00    (enemy eco pressure: 0.50)
    game_mult     : ×1.0000  (momentum:0.50 crit:0.50)
    round_mult    : ×0.80    side_mult:×1.10    save_mult:×1.000

[ARMOR]    Light Shield  (+25 HP)  400 cr

[SIDEARM]  — tidak dibutuhkan


Pengeluaran : 2,650 cr  |  Simpan: 550 cr (17%)
Save target : 800 cr  |  ⚠Violation: 250 cr (penalty: 12.50)
Utilitas    : 371.5 (adjusted: 359.0)
```

Fig 4.2 Test Case 2 Results (Source: Author's program)

For a player playing as agent Jett doing a eco round to save credits for a full buy next round, the resulting optimization are:

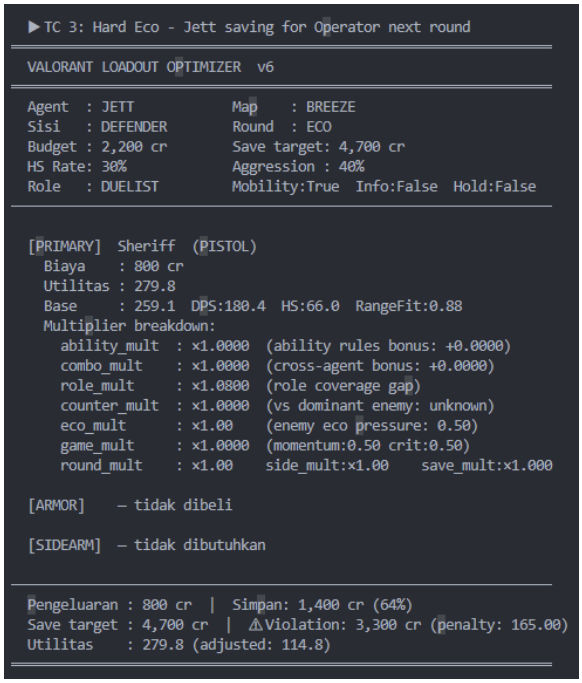


Fig 4.3 Test Case 3 Results (Source: Author's program)

B. Analysis

The resulting output shows the program successfully provides buying selection for weapon, armor, and sidearms with good accuracy. The program are able to provide descriptive results on multiplier breakdowns and synergies that happens between agents, weapons, and maps, and are able to do credit budgeting with the highest utility value without going over the credit budget that the player has. The program also successfully applies saving target as soft constraints, prioritizing achieving save target while also allowing weapon combinations that provide greater utility and combat value than achieving the save target.

However, there are several test cases that still provide soft inaccuracies in credit optimization when compared to actual Valorant game strategies. Several test cases show slight inaccuracies such as showing strategist agents that primarily fight from the back with buying shotguns that primarily needs closer range thus showing higher risk in buying strategy. These soft inaccuracies are caused by limitations on designing the program. The limitations present in the program are:

1. Heuristic values : Several of the multiplier/modifiers set in the program are based on heuristic values and are not verified by empirical evidence. Thus, the optimal solution provided by the program are based on the assumptions and constraints set on the program
2. Incomplete data: Several datasets used on this program (such as Agents and Maps) are still hardcoded and uses the most recent patch (12.11). This means any change from future patch notes and newer strategies could subject to different optimization compared to the optimal solution provided by the program. Several data are also incomplete, and the program are still using several data as test case examples.
3. Static counter table : The constraint and modifier tables implemented in the program are widely static and are not subjected for dynamic data models. This means this program are mainly used for reactive, homogeneous enemy models and no data-based adaption are implemented in the program
4. Independent modifiers : The synergy modifiers are assumed as independent and mutually exclusive, while actual game matches in Valorant could be highly dependent from several synergies. This could provide gaps and inaccuracies for more test case.
5. Missing synergies and strategies : Strategies inside Valorant are very deep with strategies changing dynamically inside matches. While map-agent-weapon synergies are mapped, there are still several synergies and strategies not taken account inside the program, such as team-wide strategy and composition and player positioning. These missing synergies could provide gaps and inaccuracies towards the optimal solution
6. Limited to Game Context : Information are limited to the player context, player stats, and game context provided on the test case. While information are sufficient enough to provide context on the game, the context are still largely static and missing several smaller heuristic values. This means the optimal solution are still constraint towards the heuristic values set on the program, and can subject to change or inaccuracies when tested on real matches.

Overall, the program are able to successfully optimize gun economy in Valorant based on the provided player and game context inside test cases with good accuracy.

V. CONCLUSION

This paper successfully implemented an optimization program for Valorant gun economy using Dynamic Programming with Knapsack-based Approach. Optimization are achieved by evaluating utility scores through heuristic measures, where total loadout scores are then calculating and compared using Dynamic Programming. The resulting solution are quite accurate in providing optimized gun selection based on player and game context alongside the buying strategy and save target that the test case or user provided.

However, there are still limitations within the program such as pure heuristic without any empirical validations and static data. These limitations can provide inaccuracies and aren't dynamic towards any change in data and the program uses less context than an actual game of Valorant. To improve optimization, future works can apply more dynamic data and context and add better heuristic measures with empirical evidence to ensure accuracy while providing better and more dynamic application. Better and updated data can also be provided and applied to the program as the program still uses dummy data with several missing agents and maps.

Overall, this paper successfully achieved optimization for gun economy in Valorant based on the heuristic and assumptions presented in the paper.

VIDEO LINK AT YOUTUBE AND SOURCE CODE

Video : https://youtu.be/GDulmJViFtk

Source code : https://github.com/Nathan-Marpaung/Application-of-Dynamic-Programming-and-Knapsack-based-Approach-in-Optimizing-Gun-Economy-in-Valorant

ACKNOWLEDGMENT

I would like to express gratefulness towards the Almighty God for providing me the power and motivation to do and finish this paper. I would also like to express gratefulness towards my parents that has been providing support from far away and help me sustain and support myself both physically and mentally until now.

Lastly, I would like to express appreciation for the lecturers of IF2211 Strategi Algoritma, specifically Mr. Rila Mandala and Mr. Rinaldi Munir for providing detailed guidance and materials through their lectures. Every token of appreciation are forwarded for the lecturers as I am able to grow and improve my knowledge regarding different types of strategy and algorithms, and I was able to apply the materials taught in lectures for this paper.

REFERENCES

[1] GeeksforGeeks, "Dynamic Programming (DP) Introduction," GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/dsa/introduction-to-dynamic-programming-data-structures-and-algorithm-tutorials/. [Accessed: 17-Jun-2026].

[2] GeeksforGeeks, "0/1 Knapsack Problem," GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/dsa/0-1-knapsack-problem-dp-10/. [Accessed: 17-Jun-2026].

[3] GeeksforGeeks, "Introduction to Knapsack Problem, Its Types and How to solve them," GeeksforGeeks. [Online]. Available" https://www.geeksforgeeks.org/dsa/introduction-to-knapsack-problem-its-types-and-how-to-solve-them/. [Accessed: 17-Jun-2026]

[4] Mobalytics, "Valorant Weapons Stats and Guides," Mobalytics. [Online]. Available: https://mobalytics.gg/valorant/weapons. [Accessed: 1-Jun-2026].

[5] R. Munir, "Program Dinamis (Bagian 1)," Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/25-Program-Dinamis-(2026)-Bagian1.pdf. [Accessed: 18-Jun-2026].

[6] H. Kellerer, U. Pferschy, and D. Pisinger, "The Multiple-Choice Knapsack Problem," in Knapsack Problems. Berlin, Germany: Springer, 2004, pp. 317–347. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-24777-7_11. [Accessed: 19-Jun-2026].

[7] USM-F, "Dynamic Programming Solution of MCKP in Python," GitHub Gist. [Online]. Available: https://gist.github.com/USM-F/1287f512de4ffb2fb852e98be1ac271d. [Accessed: 19-Jun-2026].

PERNYATAAN

Dengan ini saya menyatakan 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

Ttd

Nathan Edward Christofer Marpaung

13524062