Weapon Generator and Validator using Graph Based Model of Part Interaction in Borderlands 2

Raysha Erviandika Putra - 13524050[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]rayshaervi557@gmail.com, 13524050@std.stei.itb.ac.id

***Abstract*—Borderlands 2's signature modular weapon system assembles firearms from interchangeable parts according to compatibility and statistical rules. We introduce BLWeap, a prototype that formalizes this process as a constraint-satisfying sub-graph traversal: parts are nodes in a NetworkX graph, edges encode compatibility, and a depth-first backtracking search assembles one part per slot (body, barrel, grip, etc.) under user-defined class, manufacturer, element, and stat thresholds. In practice, BLWeap parses community-maintained dumps for realistic part data, prunes infeasible branches early, and halts on the first valid build. A Tkinter GUI with embedded Matplotlib visualizations displays both the selected parts and their induced compatibility sub-graph. BLWeap illustrates how graph theory and search algorithms can be applied to procedural content generation in interactive applications.**

***Keywords*—Borderlands 2, Graph Theory, Graph Traversal, DFS, Weapon Generation**

I. Introduction

Borderlands 2 is an RPG first-person shooter developed by Gearbox Software and released in 2012. The game is renowned for its chaotic combat, expansive loot system, and especially its procedurally generated weapons, which set it apart from other titles in the genre [1]. Unlike traditional RPG shooters that provide a fixed set of weapons with predetermined statistics, Borderlands 2 features a unique modular weapon system in which every weapon is assembled from several interchangeable parts. These components include the barrel, grip, body, stock, accessories, and elemental modules, each contributing distinct mechanical and visual characteristics to the final weapon.

Weapons in Borderlands 2 are categorized into several classes, such as pistols, submachine guns (SMGs), shotguns, sniper rifles, and assault rifles. Each class is defined by its own set of compatible part types and visual design rules. Additionally, weapon parts are associated with many fictional manufacturers, including Jakobs, Maliwan, Hyperion, and others, each imparting unique stylistic elements and statistical tendencies to the weapon. The modular nature of this system means that each part contributes specific statistical effects such as damage, accuracy, fire rate, reload speed, and magazine size as well as visual aesthetics, resulting in millions of possible weapon combinations and a highly diverse loot experience .

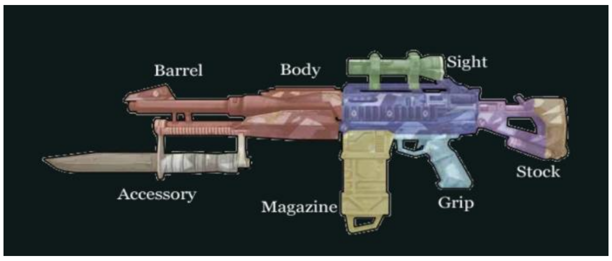


Fig 1 Borderlands 2 Parts Visual

Source https://borderlands-archive.fandom.com/wiki/Borderlands_2:Weapon_Components

In this paper, we introduce BLWeap, a system designed to model Borderlands 2's weapon generation using concepts from graph theory and constraint satisfaction. BLWeap encodes the modular weapon system as a structured graph of components and their interrelations. By mapping out which part can coexist and how they interact, then performs a guided search through this graph, respecting threshold, to assemble only those combinations that meet all criteria specified by the user. This formalization enables the system to generate valid weapon configurations under a range of constraints, such as manufacturer preferences, required part types, or minimum statistical thresholds.

To ensure that the system accurately reflects the game's logic, structured part definitions were extracted from the GitHub repository gibbed/Borderlands2Dumps [2], maintained by the community. This dataset provides access to raw part identifiers, classifications, and compatibility information directly from the game files, enabling the construction of a graph-based model that mirrors the actual data and constraints found in Borderlands 2.

II. Theory Foundation

A. Weapon Rules

Weapon Generation Rules Weapon generation in Borderlands 2 is controlled by several rules [3] that ensure both the validity and diversity of the generated weapons:

1. Class Consistency: All parts used to construct a weapon must belong to the same weapon class. The primary weapon classes in Borderlands 2 are Pistols, Submachine Guns (SMGs), Shotguns, Sniper Rifles, and Assault Rifles. Each class has its own set of compatible parts.
2. Stat Modifiers: Every weapon part contributes specific statistical modifiers such as damage, accuracy, fire rate,

reload speed, magazine size and other stats which collectively determine the final attributes of the assembled weapon.

3. Manufacturer Synergies: Certain parts provide additional bonuses when combined with other parts from the same manufacturer. For example, equipping a Jakobs grip on a weapon with Jakobs manufacturer will grant increased magazine size or faster reload speed. The game features several fictional manufacturers, including Jakobs, Bandit, DAHL, Hyperion, Maliwan, Tediore, Torgue, and Vladof.

4. Part-Type Uniqueness: A valid weapon must include exactly one part of each required type (e.g., one barrel, one grip, one body, etc.), the number of part needed varies between weapon class.

5. Manufacturer Determination: The manufacturer of a weapon is determined by the body part used in its construction, influencing both the weapon's aesthetics and its core statistical tendencies, whilst the part other than body can merge having multiple different manufacturer sources.

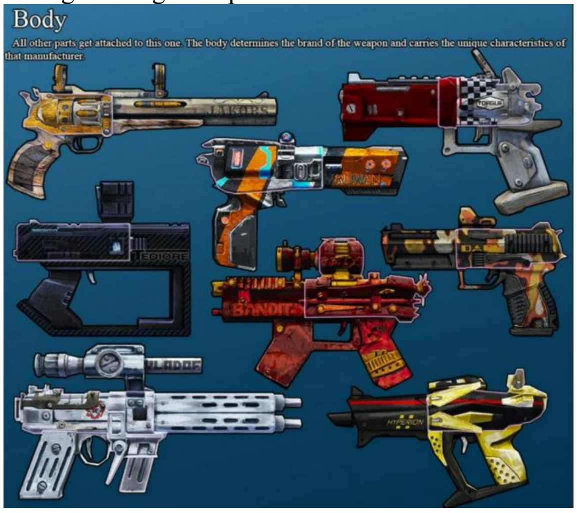

Body
All other parts get attached to this one. The body determines the brand of the weapon and carries the unique characteristics of that manufacturer.
JAKOBS
TEDIORE
BANDIT
VLADOF
HYPERION

Fig 2 Body Part of weapon class "Pistol"

Source https://imgur.com/gallery/borderlands-2-weapon-parts-wkkWi

Fig 2 shows all pistol class weapon manufacturer with their corresponding unique weapon parts (Maliwan with Maliwan parts, Bandit with Bandit parts, etc). and here are images showing each versions of weapon class

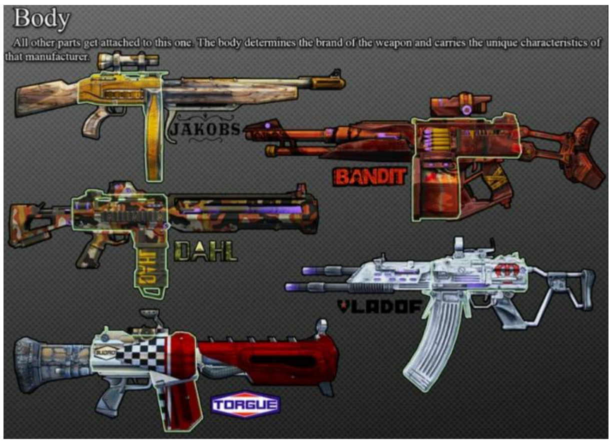

Body
All other parts get attached to this one. The body determines the brand of the weapon and carries the unique characteristics of that manufacturer.
JAKOBS
BANDIT
DAHL
VLADOF
TORGUE

Fig 3 Body Parts of weapon class "Assault Rifles"

Source https://imgur.com/gallery/borderlands-2-weapon-parts-wkkWi

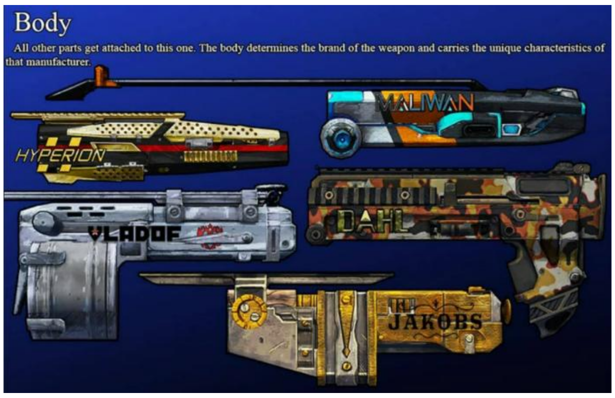

Body
All other parts get attached to this one. The body determines the brand of the weapon and carries the unique characteristics of that manufacturer.
MALIWAN
HYPERION
VLADOF
DAHL
JAKOBS

Fig 4 Body Parts of weapon class "Sniper"

Source https://imgur.com/gallery/borderlands-2-weapon-parts-wkkWi

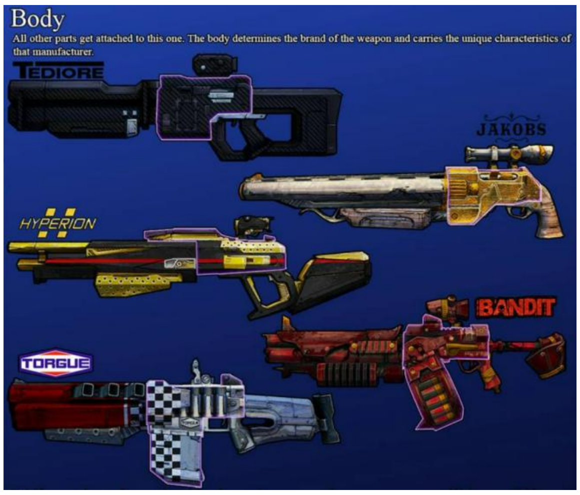

Body
All other parts get attached to this one. The body determines the brand of the weapon and carries the unique characteristics of that manufacturer.
TEDIORE
JAKOBS
HYPERION
BANDIT
TORGUE

Fig 5 Body Parts of weapon class "Shotgun"

Source https://imgur.com/gallery/borderlands-2-weapon-parts-wkkWi

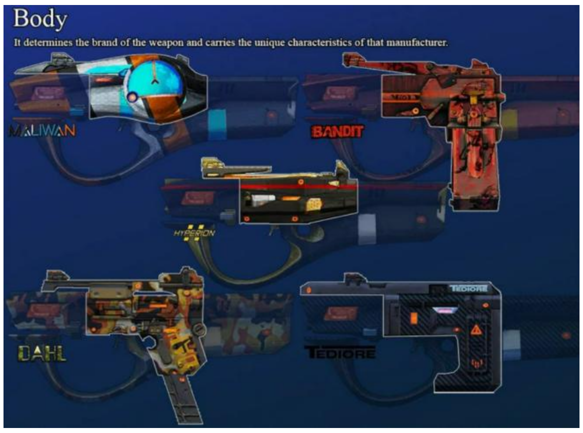

Body
It determines the brand of the weapon and carries the unique characteristics of that manufacturer.
MALIWAN
BANDIT
HYPERION
DAHL
TEDIORE

Fig 6 Body Parts of weapon class "SMG"

Source https://imgur.com/gallery/borderlands-2-weapon-parts-wkkWi

Shown from fig 2 through fig 6, each combination of fully built weapon with distinctive body parts for each weapon class and manufacturer, highlighting how visual style and part-type rules vary across classes. In the next section, we translate each possible part into a unique item or object.

B. Graph

Graph G is defined as a pair $G = (V, E)$, where V is a set of vertices or nodes and E is a set of edges connecting pairs of vertices [4]. In this context, nodes: Represent individual weapon parts where each weapon parts has attributes: id (as specifier), Type, Class, Manufacturer, Stat Modifiers, and Compatibility. Edges: Represent compatibility between parts. Edge (A, B) exists if part A can be combined with part B.

a. Undirected Graph

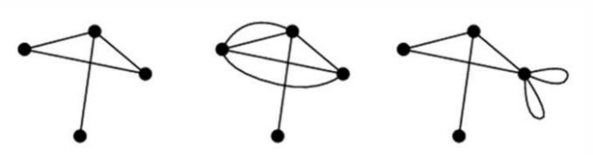


Fig 6 Undirected Graph

Source [*https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025*](https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025)

An undirected graph is a collection of nodes connected by edges that have no arrowheads. each edge simply indicates a two-way relationship. In the example fig 6: Simple undirected graph Every pair of connected vertices has exactly one edge and there are no loops or parallel edges. Undirected multigraph. Some vertex pairs are joined by multiple edges, illustrating that more than one "connection" can exist between the same two nodes.

b. Adjacent

Two vertices u and v in G are said to be adjacent if there exists an edge $(u, v) \in E$ connecting them [4]. In the context of weapon construction, adjacency means the two parts can be combined in a single weapon according to game constraints (such as same weapon class and allowed combinations). For example, a Jakobs Barrel Pistol and Jakobs Grip Pistol are adjacent if they can be used together in a pistol build.

c. Subgraphs

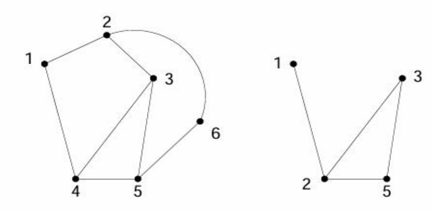


Fig 7 Subgraph

Source [*https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025*](https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025)

A subgraph $H = (V', E')$ of G is a graph where $V' \subseteq V$ and $E' \subseteq E$ [4]. In this paper, a fully built weapon can be represented as a graph of parts, the system constructs a sub-graph consisting of the relevant parts to make a weapon, each containing exactly one part of each required type.

d. Complete Graph

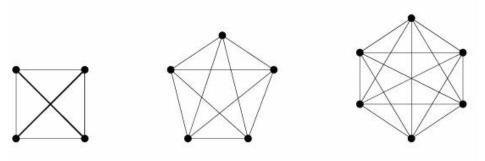


Fig 8 Complete Graphs

Source [*https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025*](https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025)

A complete graph is a simple graph in which every vertex has an edge to every other vertex. A complete graph on n vertices is denoted K_n. The number of edges in a complete graph with n vertices is $n(n-1)/2$ [4]. In a weapon build, all parts must be connected with each other that ensures compatibility between all part.

e. Hamiltonian Graph

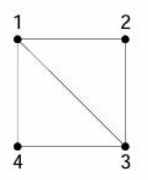


Fig 9 Hamiltonian Graph

Source [*https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025*](https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025)

A Hamiltonian circuit in a graph is a sequence of adjacent vertices. A Hamiltonian circuit is a circuit that visits each vertex exactly once and is able to return to the origin vertex [4], example on fig 8 where traversal goes from 1, 2, 3, 4. In weapon generation, a valid build corresponds to a Hamiltonian circuit (or sub-graph) that includes one part from each required type, ensuring all parts are mutually compatible and the build is complete. This process also there is more part left that requires checking as it traces back to origin part.

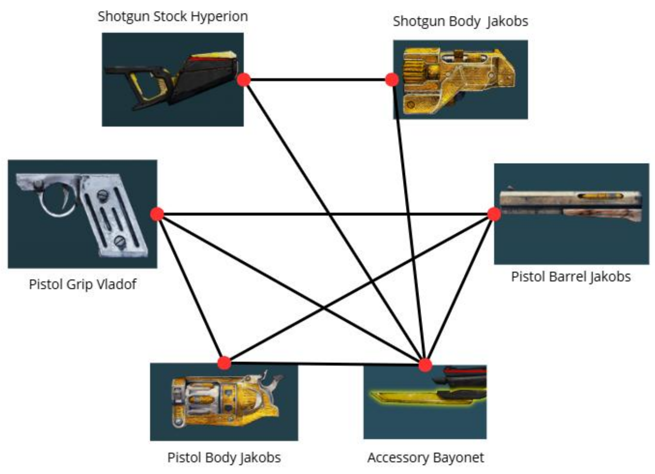


Fig 10 Sub-graph Example Scenario

An example of constructed graph showed in fig 6 where the resulted weapon would be a Pistol with the manufacturer Jakobs, Vladof Grip, Jakobs Barrel, and a Bayonet Accessory. The graph shows parts compatibility between class pistol and shotgun (as example), the four different parts of the pistol is able to make a Hamiltonian circuit whilst also able to satisfy the minimum parts needed to construct a weapon. A special case towards accessories which is a rather universal part where some accessories is applicable to all weapon class (bayonet as example). Shotgun parts is able to connect to each other and at the same time connect to the bayonet accessory, although the shotgun is able to make a Hamiltonian circuit, it does not satisfy the minimum parts needed to construct a weapon.

C. Search-Based Selection

The weapon generation itself is modeled as a constraint-satisfying sub-graph traversal problem. The process of validity checking in the system is equivalent to searching for a Hamiltonian path under constraints, where: Each node in the path must represent a distinct type, all selected nodes must be pairwise connected (i.e., compatible) Additional user-defined constraints (e.g., minimum damage, specific manufacturer) must be satisfied.

1. Starting Node: Manufacturer Body (Anchor). Since Borderlands 2 determines the weapon's manufacturer primarily through its body, the generator begins by selecting candidate body parts that match any specified manufacturer threshold. This aligns with a rooted search where: The root node R is the selected body part. From the root, the search will branch to other possible part.

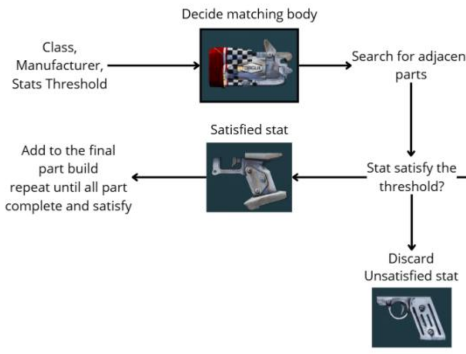

Decide matching body
Class, Manufacturer, Stats Threshold
Search for adjacent parts
Satisfied stat
Add to the final part build repeat until all part complete and satisfy
Stat satisfy the threshold?
Discard Unsatisfied stat

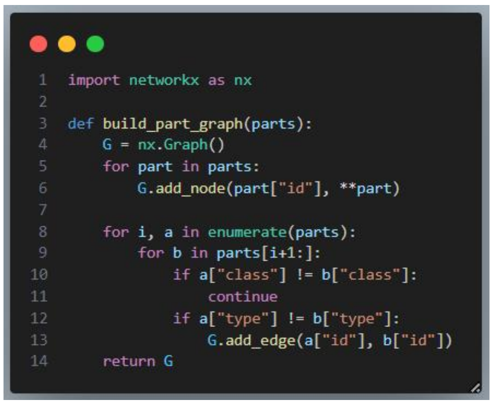
Fig 11 Generation Process Diagram

2. Search Algorithm: Depth-first search (DFS) is a graph traversal algorithm that begins at a designated root and explores as far as possible along each branch before backtracking [5]. DFS excels at constraint satisfaction tasks because it incrementally builds partial candidate solutions and applies compatibility or constraint checks at each step, pruning any branch that cannot lead to a valid solution as soon as a part that does not satisfy is detected [6]. In the weapon generation scenario, each recursion level corresponds to assigning a part (body, barrel, grip, etc.), compatibility tests eliminate infeasible builds, and the algorithm stops immediately upon finding the first fully valid configuration, avoiding full enumeration of all combinations and keeping memory usage proportional to the assignment depth rather than the total number of parts.

III. Implementation

A. Creating The Graph

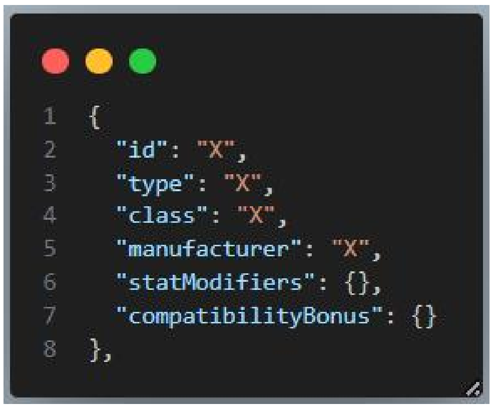
With material and resources from [5][6][7]. First a parsed JSON file containing all the weapon data from github gibbed/Borderlands2Dumps [2] is used to provide realistic in-game data. Using networkx a graph is created under the previously stated rules.



```python
import networkx as nx

def build_part_graph(parts):
    G = nx.Graph()
    for part in parts:
        G.add_node(part["id"], **part)

    for i, a in enumerate(parts):
        for b in parts[i+1:]:
            if a["class"] != b["class"]:
                continue
            if a["type"] != b["type"]:
                G.add_edge(a["id"], b["id"])
    return G
```

Fig 12 Python Code - Graph Creation

Each id in the JSON uniquely identifies a single weapon part, serving as a primary key within the system's part registry. Every part is represented as a structured JSON object, encapsulating a variety of attributes. Specifically, each part entry contains: id, type, class, and manufacturer. In addition, each part includes a statModifiers dictionary that details the numerical effects the part imparts on weapon statistics, and other stats that is not visible, as well as an optional compatibilityBonus field

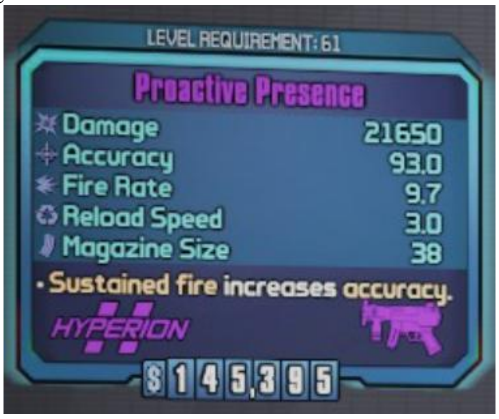


```
{
  "id": "X",
  "type": "X",
  "class": "X",
  "manufacturer": "X",
  "statModifiers": {},
  "compatibilityBonus": {}
},
```

Fig 13 JSON Object Format

B. Filtering

After the graph is created, user can enter some of the generic stats modifier for a weapon. A normal weapon card at the very least contains Damage, Accuracy, Fire Rate, Reload Speed, and Magazine Size



LEVEL REQUIREMENT: 61

Proactive Presence

Damage	21650
Accuracy	93.0
Fire Rate	9.7
Reload Speed	3.0
Magazine Size	38

• Sustained fire increases accuracy.

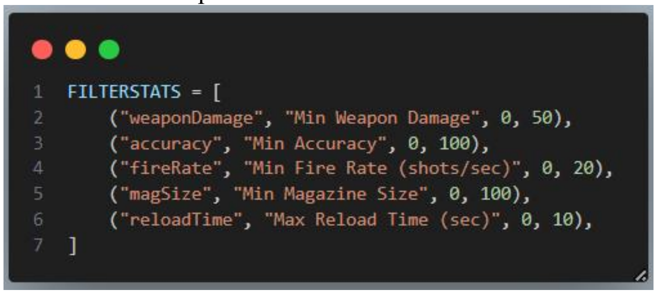
HYPERION

$145,395

Fig 14 Borderlands 2 Weapon Card

Source Screenshot Taken In-game by Author

With the limitation of the visible stats, user can state some of the stat on the weapon and set them as threshold for the weapon generation. The damage stat is calculated differently from the other stats as it accounts for level of the player, for proof of concept purposes only, the system will only accept base stat of the weapon.

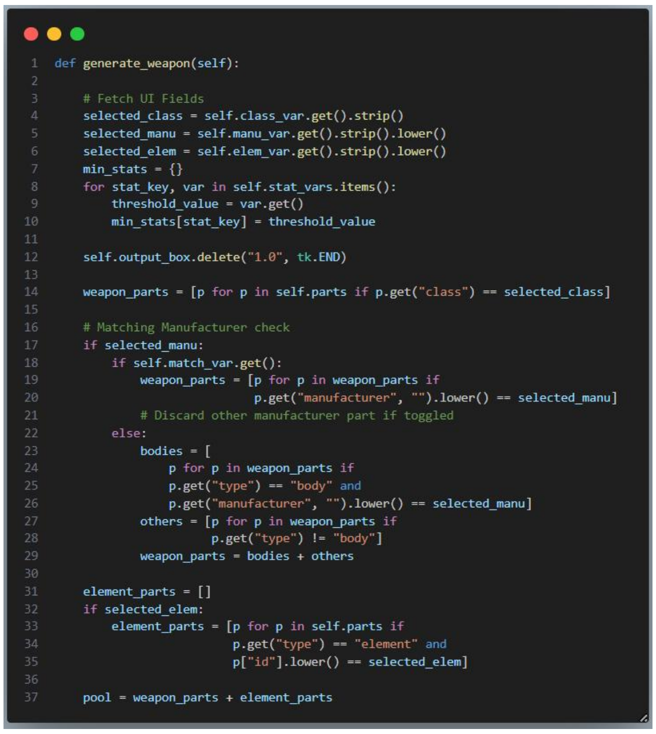


```python
FILTERSTATS = [
    ("weaponDamage", "Min Weapon Damage", 0, 50),
    ("accuracy", "Min Accuracy", 0, 100),
    ("fireRate", "Min Fire Rate (shots/sec)", 0, 20),
    ("magSize", "Min Magazine Size", 0, 100),
    ("reloadTime", "Max Reload Time (sec)", 0, 10),
]
```

Fig 15 Stat Filtering

Stat filters is mapped like above, to be able to match the fields within the JSON files, focusing on visible stats.

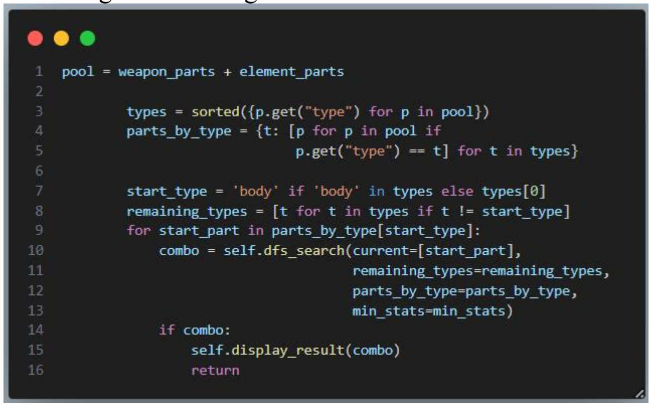
C. Weapon Generation



```python
def generate_weapon(self):

    # Fetch UI Fields
    selected_class = self.class_var.get().strip()
    selected_manu = self.manu_var.get().strip().lower()
    selected_elem = self.elem_var.get().strip().lower()
    min_stats = {}
    for stat_key, var in self.stat_vars.items():
        threshold_value = var.get()
        min_stats[stat_key] = threshold_value

    self.output_box.delete("1.0", tk.END)

    weapon_parts = [p for p in self.parts if p.get("class") == selected_class]

    # Matching Manufacturer check
    if selected_manu:
        if self.match_var.get():
            weapon_parts = [p for p in weapon_parts if
                            p.get("manufacturer", "").lower() == selected_manu]
            # Discard other manufacturer part if toggled
        else:
            bodies = [
                p for p in weapon_parts if
                p.get("type") == "body" and
                p.get("manufacturer", "").lower() == selected_manu]
            others = [p for p in weapon_parts if
                      p.get("type") != "body"]
            weapon_parts = bodies + others

    element_parts = []
    if selected_elem:
        element_parts = [p for p in self.parts if
                         p.get("type") == "element" and
                         p["id"].lower() == selected_elem]

    pool = weapon_parts + element_parts
```

Fig 36 Generate Weapon Function

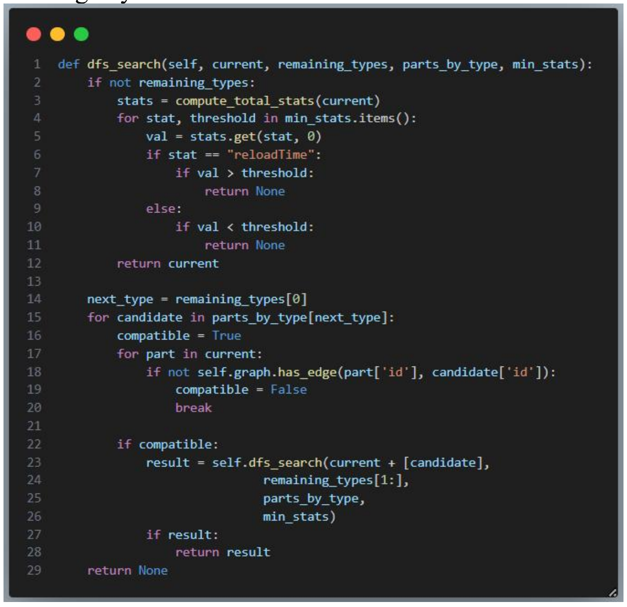
1. Fetch Value: first the function fetch fields from the UI input-ed by the user. Includes class, manufacturer, element and min_stat.
2. Filter by Class: next filtering the complete parts list down to only those matching the chosen weapon class.
3. Manufacturer Filtering :
Manufacturer Filtering Strict Mode: if the "match manufacturer" checkbox is checked, we discard every part whose manufacturer doesn't match the body's manufacturer mimicking full-brand synergy bonuses.
Body-Only Mode: If unchecked, we only force the body part to match the selected maker, then allow any manufacturer for barrels, grips, etc. This captures the game's design where the body sets the weapon's brand, but accessories can come from elsewhere.
4. Filtering element: discard element that do not match the threshold.
5. Global Pool: create a global part pool containing parts that goes through the filtering



```python
pool = weapon_parts + element_parts

        types = sorted({p.get("type") for p in pool})
        parts_by_type = {t: [p for p in pool if
                             p.get("type") == t] for t in types}

        start_type = 'body' if 'body' in types else types[0]
        remaining_types = [t for t in types if t != start_type]
        for start_part in parts_by_type[start_type]:
            combo = self.dfs_search(current=[start_part],
                                    remaining_types=remaining_types,
                                    parts_by_type=parts_by_type,
                                    min_stats=min_stats)
            if combo:
                self.display_result(combo)
                return
```

Fig 17 Type Extraction and DFS Call

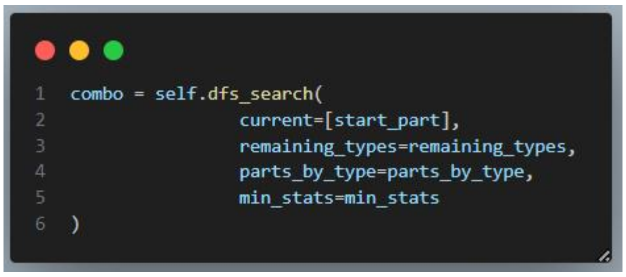
6. Type Extraction: the pool will then enter type extraction grouping types becomes a sorted list of every unique slot name (e.g. ["body", "barrel", "grip", …]). parts_by_type is a dict mapping each slot to the list of candidate parts for that slot.
7. DFS Call: Iterate each candidate for the starting slot. Call dfs_search, seeding it with the single chosen part in current and asking it to fill the rest. On the first non-None return (i.e. a valid full build), immediately call display_result and exit, preventing any further search.



```python
def dfs_search(self, current, remaining_types, parts_by_type, min_stats):
    if not remaining_types:
        stats = compute_total_stats(current)
        for stat, threshold in min_stats.items():
            val = stats.get(stat, 0)
            if stat == "reloadTime":
                if val > threshold:
                    return None
            else:
                if val < threshold:
                    return None
        return current

    next_type = remaining_types[0]
    for candidate in parts_by_type[next_type]:
        compatible = True
        for part in current:
            if not self.graph.has_edge(part['id'], candidate['id']):
                compatible = False
                break

        if compatible:
            result = self.dfs_search(current + [candidate],
                                     remaining_types[1:],
                                     parts_by_type,
                                     min_stats)
            if result:
                return result
    return None
```

Fig 18 DFS Implementation

The weapon generation system employs a depth-first search (DFS) with backtracking to assemble a complete weapon from a compatibility graph $G = (V, E)$. Given an ordered list of required part types $P = [p1, \dots, pn]$, a dictionary parts_by_type mapping each p_i to its candidate set, and a threshold map min_stats provided from FILTERSTATS

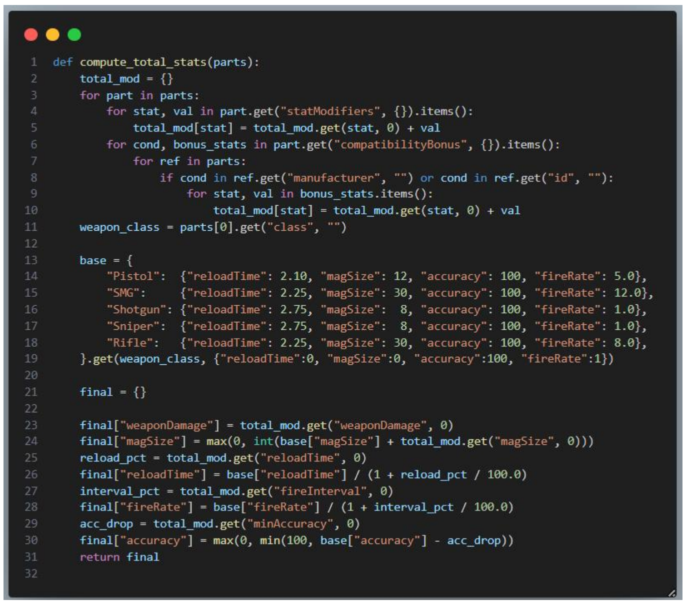


```
combo = self.dfs_search(
                current=[start_part],
                remaining_types=remaining_types,
                parts_by_type=parts_by_type,
                min_stats=min_stats
)
```

Fig 19 DFS Usage to Acquire Combo

First half of the dfs_search serves as the base case check. When no types remain, the current sequence of parts current=[p1,…,pk] forms a full build. Its total statistics are computed by summing individual statModifiers with compute_total_stat(). Each stat is checked against min_stats: for most stats a lower-bound test (value < threshold) is applied, whereas for "reloadTime" an upper-bound test (value > threshold) is applied. If all constraints pass, the sequence is returned, otherwise the branch is abandoned.

Recursive is done by the second half of the code, next_type is the first element of the remaining_types list. For each candidate part $c \in$ parts_by_type[next_type], a compatibility check ensures that the two parts has edge between them, $(c, p) \in E$ for every $p \in$ current. Incompatible candidates are discarded immediately. Compatible candidates trigger a recursive call with c appended to current and next_type removed from the remaining_type list. If any recursive call yields a non-None result, it is propagated upward without exploring further siblings, thus implementing a first-solution exit.



```
def compute_total_stats(parts):
    total_mod = {}
    for part in parts:
        for stat, val in part.get("statModifiers", {}).items():
            total_mod[stat] = total_mod.get(stat, 0) + val
        for cond, bonus_stats in part.get("compatibilityBonus", {}).items():
            for ref in parts:
                if cond in ref.get("manufacturer", "") or cond in ref.get("id", ""):
                  for stat, val in bonus_stats.items():
                      total_mod[stat] = total_mod.get(stat, 0) + val
    weapon_class = parts[0].get("class", "")

    base = {
        "Pistol":  {"reloadTime": 2.10, "magSize": 12, "accuracy": 100, "fireRate": 5.0},
        "SMG":     {"reloadTime": 2.25, "magSize": 30, "accuracy": 100, "fireRate": 12.0},
        "Shotgun": {"reloadTime": 2.75, "magSize":  8, "accuracy": 100, "fireRate": 1.0},
        "Sniper":  {"reloadTime": 2.75, "magSize":  8, "accuracy": 100, "fireRate": 1.0},
        "Rifle":   {"reloadTime": 2.25, "magSize": 30, "accuracy": 100, "fireRate": 8.0},
    }.get(weapon_class, {"reloadTime":0, "magSize":0, "accuracy":100, "fireRate":1})

    final = {}

    final["weaponDamage"] = total_mod.get("weaponDamage", 0)
    final["magSize"] = max(0, int(base["magSize"] + total_mod.get("magSize", 0)))
    reload_pct = total_mod.get("reloadTime", 0)
    final["reloadTime"] = base["reloadTime"] / (1 + reload_pct / 100.0)
    interval_pct = total_mod.get("fireInterval", 0)
    final["fireRate"] = base["fireRate"] / (1 + interval_pct / 100.0)
    acc_drop = total_mod.get("minAccuracy", 0)
    final["accuracy"] = max(0, min(100, base["accuracy"] - acc_drop))
    return final
```

Fig 20 Stat Calculation Code

Stat calculation is done by iterating through the part's stat modifier entry, and it adds the modifier value to total_mod[stat]. Next, it applies any "compatibilityBonus" entries, for each bonus rule keyed by a condition, it checks if that condition is true in the current build, if so it adds bonus stat to total_mod[stat] as well.

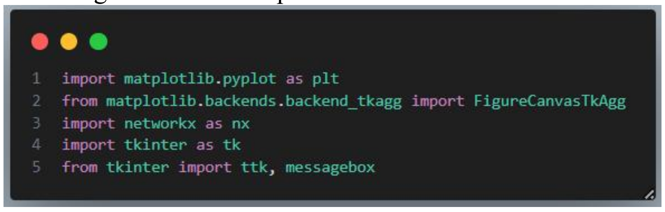
Hard coded values for the base stat mimics a simplified version of the stat calculation of the weapon, since most of the base stat is affected by other stat outside the weapon scope. The base field represent the unmodified in-game stats for that weapon body.

The final stat calculation is counted through some formulation [2] such as:

1. $\text{Damage} = \text{Weapon Damage} + \text{total\_mod[damage]}$ (1)
2. $\text{Mag} = \max(0, \text{basemagSize} + \text{total\_mod[magSize]})$ (2)
3. $$T_{\text{reload}} = \frac{B_{\text{reloadTime}}}{1 + \dfrac{\text{total\_mod(reloadTime)}}{100}} \quad (3)$$
4. $$R_{\text{fire}} = \frac{B_{\text{fireRate}}}{1 + \dfrac{\text{total\_mod(fireInterval)}}{100}} \quad (4)$$
5. $$Accu = \min\Big(100,\ \max\big(0,\ B_{\text{accuracy}} - \text{total\_mod(minAccuracy)}\big)\Big) \quad (5)$$

D. Visualization

NetworkX, Matplotlib, and tkinter is used, After a valid weapon is found, then extract just those parts from the full compatibility graph and render them as a small network diagram. A layout spaces the nodes, which are color coded by part type (body, barrel, grip, etc.). Edges between nodes show compatibility, and each node is labeled with its part ID. This Matplotlib figure is then embedded directly into the Tkinter GUI alongside the text output.

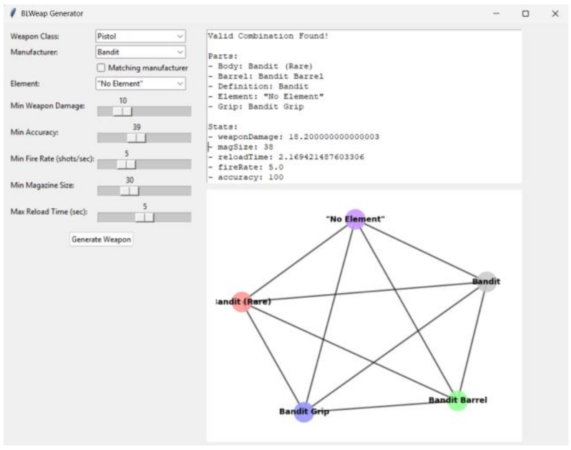


```
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import networkx as nx
import tkinter as tk
from tkinter import ttk, messagebox
```

Fig 21 Library Imports for Visualization

IV. Results

The implementation of this paper resulted a python app that can be run, user can enter threshold from the drop down or the sliders, and the generate weapon button will show the generated weapon result.



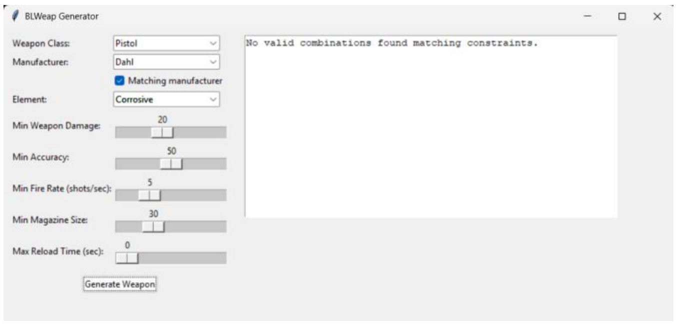
BLWeap Generator

Weapon Class: Pistol
Manufacturer: Bandit
☐ Matching manufacturer
Element: "No Element"
Min Weapon Damage: 10
Min Accuracy: 39
Min Fire Rate (shots/sec): 5
Min Magazine Size: 30
Max Reload Time (sec): 5

```
Valid Combination Found!

Parts:
- Body: Bandit (Rare)
- Barrel: Bandit Barrel
- Definition: Bandit
- Element: "No Element"
- Grip: Bandit Grip

Stats:
- weaponDamage: 18.200000000000003
- magSize: 38
- reloadTime: 2.169421487603306
- fireRate: 5.0
- accuracy: 100
```

Fig 22 Result of Build 1

Build 1 (Fig. 22) demonstrates that, for modest thresholds (Pistol, Bandit, no element, mid-range stats), the generator finds a body, barrel, grip, stock, and accessory that all share compatibility edges. This confirms that our graph model faithfully encodes in-game part compatibilities.

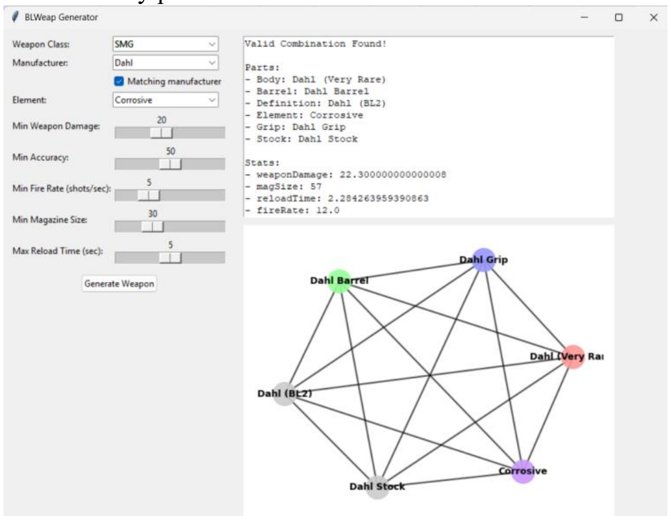


Weapon Class: Pistol
Manufacturer: Dahl
[✓ Matching manufacturer]
Element: Corrosive
Min Weapon Damage: 20
Min Accuracy: 50
Min Fire Rate (shots/sec): 5
Min Magazine Size: 30
Max Reload Time (sec): 0

```
No valid combinations found matching constraints.
```

Fig 23 Result of Build 2 (No Satisfied Part)

Build 2 (Fig. 23) shows a “no valid combination” case when thresholds are impossibly strict (e.g., reloadTime = 0). This aligns with game logic aside from the special “Infinity Pistol,” no build can meet zero reload time, indicating our threshold checks correctly prune all branches.

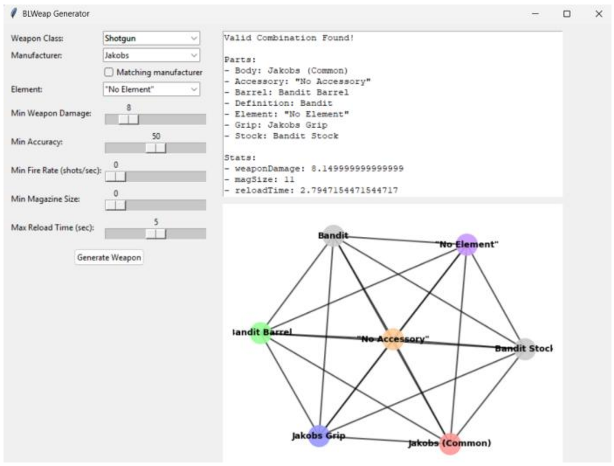


Weapon Class: SMG
Manufacturer: Dahl
[✓ Matching manufacturer]
Element: Corrosive
Min Weapon Damage: 20
Min Accuracy: 50
Min Fire Rate (shots/sec): 5
Min Magazine Size: 30
Max Reload Time (sec): 5

```
Valid Combination Found!

Parts:
- Body: Dahl (Very Rare)
- Barrel: Dahl Barrel
- Definition: Dahl (BL2)
- Element: Corrosive
- Grip: Dahl Grip
- Stock: Dahl Stock

Stats:
- weaponDamage: 22.300000000000008
- magSize: 57
- reloadTime: 2.284263959390863
- fireRate: 12.0
```

Fig 24 Build Result 3 (Matching Manufacturer)

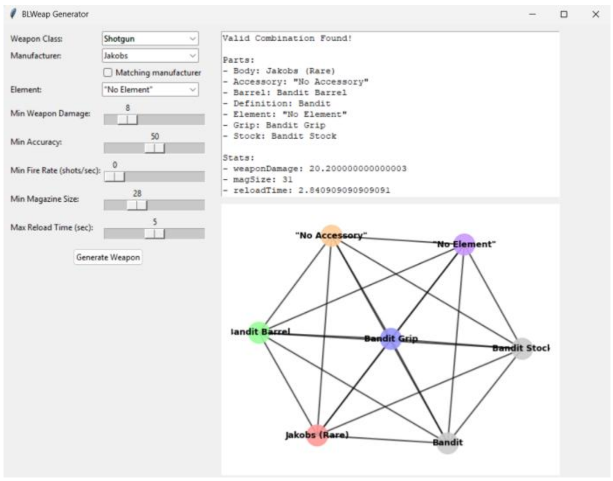
Build 3 (Fig. 24) highlights the effect of the “Matching manufacturer” option. By forcing all non-body parts to share the body’s maker, we exploit in-game synergy bonuses. Notice the chosen grip and barrel come from the same manufacturer, improving stats at the cost of reducing the search space. This shows our model accommodates secondary “compatibility bonus” rules without additional code complexity.



Weapon Class: Shotgun
Manufacturer: Jakobs
Matching manufacturer
Element: "No Element"
Min Weapon Damage: 8
Min Accuracy: 50
Min Fire Rate (shots/sec): 0
Min Magazine Size: 0
Max Reload Time (sec): 5

```
Valid Combination Found!

Parts:
- Body: Jakobs (Common)
- Accessory: "No Accessory"
- Barrel: Bandit Barrel
- Definition: Bandit
- Element: "No Element"
- Grip: Jakobs Grip
- Stock: Bandit Stock

Stats:
- weaponDamage: 8.149999999999999
- magSize: 11
- reloadTime: 2.7947154471544717
```

Figure 25 Result Build 4 (Lower Threshold Result)



Weapon Class: Shotgun
Manufacturer: Jakobs
Matching manufacturer
Element: "No Element"
Min Weapon Damage: 8
Min Accuracy: 50
Min Fire Rate (shots/sec): 0
Min Magazine Size: 28
Max Reload Time (sec): 5

```
Valid Combination Found!

Parts:
- Body: Jakobs (Rare)
- Accessory: "No Accessory"
- Barrel: Bandit Barrel
- Definition: Bandit
- Element: "No Element"
- Grip: Bandit Grip
- Stock: Bandit Stock

Stats:
- weaponDamage: 20.200000000000003
- magSize: 31
- reloadTime: 2.840909090909091
```

Figure 26 Result Build 5 (Higher Threshold Result)

Build 4 (Fig. 25) and Build 5 (Fig. 26) illustrate how raising the magazine-size minimum causes the algorithm to switch from a Jakobs to a Bandit grip. Bandit parts are known for flat mag-size bonuses, so the system naturally gravitates toward them under tighter constraints. Thus, our generator not only respects thresholds but also reflects known manufacturer tendencies.

The generator currently ignores rare elemental procs, secondary bonuses (e.g., elemental overcharge), and level scaling. This can lead to “false negatives” where, in the real game engine, a high-level Maliwan barrel would meet thresholds via hidden multipliers. In edge cases where JSON dumps omit certain parts (e.g., unreleased DLC components), the system reports “no valid combination” correctly, but without distinguishing “impossible” from “data missing.” A future data-validation pass could surface these gaps.

Overall for simplified version of the generation system, the model provide accurate results, outside that matter, exist in-game factors that is not stated explicitly within the constraints or JSON file itself.

V. Conclusion

Borderlands 2 Weapon Generation system is a unique mechanic yet lacks formal documentation, BLWeap is a prototype tool that turns Borderlands 2's modular weapon system into a graph-based search. It pulls part data from the gibbed dumps, builds a compatibility graph in NetworkX, and uses a DFS backtracking algorithm to generate weapons that satisfy user-defined thresholds (damage, accuracy, fire rate, magazine size, reload time) Validate part combinations against both compatibility and stat rules Visualize each successful build as a color-coded subgraph. Future work could add advanced stat models, multi-objective optimization, richer GUI filters, batch generation, direct integration into a Borderlands 2 mod, and weapon visuals for each part to represent the final weapon appearance. BLWeap shows how graph theory and search algorithm can drive procedural content generation in games.

VI. Appendix

Implementation of BLWeap above is stored within the github. https://github.com/Arbane557/BLWeap Data used for the implementation of BLWeap is provided from a github https://github.com/gibbed/Borderlands2Dumps

VII. Acknowledgment

The author extends sincere appreciation to Arrival Dwi Sentosa, S.Kom., M.T., whose invaluable guidance, support, and encouragement throughout the preparation of this manuscript have been truly instrumental. His expertise, insightful feedback, and unwavering dedication to teaching have greatly enriched the author's understanding of the subject matter.

References

[1] IGN, "Borderlands 2," 2012. [Online]. Available: https://www.ign.com/games/borderlands-2 [Accessed 18 June 2025].

[2] "gibbed/Borderlands2Dumps," GitHub, 2014. [Online]. Available: https://github.com/gibbed/Borderlands2Dumps [Accessed 19 June 2025].

[3] BL2.Parts, "Borderlands 2 Parts Database." [Online]. Available: https://bl2.parts/ [Accessed 19 June 2025]

[4] R. Munir, "Matdis Course Materials (2024–2025)," Dept. Informatika, STEI-ITB. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/

[5] GeeksforGeeks, "Depth-first Search or DFS for a Graph," 2019. [Online]. Available: https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/ [Accessed 19 June 2025].

[6] GeeksforGeeks, "Explain the Concept of Backtracking Search and its Role in Finding Solutions to CSPs," 2021. [Online]. Available: https://www.geeksforgeeks.org/explain-the-concept-of-backtracking-search-and-its-role-in-finding-solutions-to-csps/ [Accessed 19 June 2025].

[7] Matplotlib Development Team, "Matplotlib Tutorial," 2025. [Online]. Available: https://matplotlib.org/stable/tutorials/ [Accessed 20 June 2025].

[8] Togelius, J., Shaker, N., & Nelson, M. J. (Eds.). (2016). Procedural Content Generation in Games. Springer.

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, 20 Juni 2025

[illegible]

Raysha Erviandika Putra 13524050