Graph-Based Modeling of osu! Beatmaps for Estimating Gameplay Difficulty

Emilio Justin – 13524043
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: *emilio.justin54@gmail.com*, *13524043@std.stei.itb.ac.id*

***Abstract*— osu! is a free-to-play rhythm game where beatmaps are designed as sequences of interactive hit objects synchronized to music. While the game provides a star rating system to indicate a beatmap difficulty, the underlying calculation is not easily interpretable. This paper presents an alternative approach to estimate a beatmap difficulty by modeling osu! beatmaps as directed weighted graphs, where each node represents a hit object and edges represents transitions based on spatial and temporal relationships. A method is developed for parsing beatmap files, constructing the corresponding graph, and evaluating the resulting model using a custom edge-weighting formula that reflects movement difficulty. This graph-based representation offers an interpretable framework to estimate its difficulty.**

***Keywords* — osu!, Graph, Beatmap Modeling, Difficulty Estimation**

I. Introduction

Osu! is a free-to-play rhythm game in which *hit objects* appear in sync with the music, and players are required to click, slide, or spin these objects at the right time and in the correct sequence. Each beatmap in osu! represents a unique pattern of these hit objects, carefully designed to match the tempo and structure of the accompanying song. The gameplay emphasizes not only timing precision, but also spatial awareness and quick decision-making, making it both mechanically demanding and cognitively engaging.

The difficulty of an osu! beatmap is influenced by multiple factors, including the density of hit objects, their spatial distribution, the approach rate of the hit objects, and the complexity of movement patterns between them. While osu! provides a star rating system to represent the overall difficulty of a beatmap, the underlying calculation involves a combination of heuristics and simulation-based estimations that are not easily accessible or interpretable.

This paper presents an alternative approach to estimate a beatmap difficulty using graph theory as a modeling framework. By parsing an osu! beatmap file and constructing a weighted directed graph where nodes represent hit objects and edges capture spatial and temporal relationships between them, with each edge is weighted using a custom formula that reflects the difficulty of transitioning between two objects. This graph-based representation is intended to provide a more structured, transparent, and interpretable method for estimating gameplay difficulty in osu! beatmaps.

II. Theoretical Framework

A. Graph

In mathematics, a graph is a fundamental structure used to model relations between discrete objects. A graph consists of two primary components, a set of vertices (also called nodes) and a set of edges that connect pairs of vertices.

1) Definition
Formally, a graph can be defined as an ordered pair $G = (V, E)$, where:
- V is a non-empty set of vertices, such that $V = \{v_1, v_2, ..., v_n\}$
- E is a set of edges, where each edge represents a connection between two vertices, such that $E = \{e_1, e_2, ..., e_n\}$

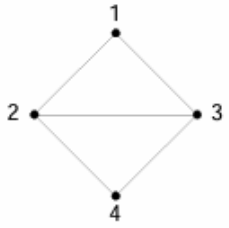

Fig. 1: Simple Graph
source: https://informatika.stei.itb.ac.id/ rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf

2) Types of Graphs
Graphs can be classified into several types based on their connections and the presence of weights on their edges. In an undirected graph, the edges have no direction, meaning that the connection between nodes is mutual. In contrast, a directed graph includes edges with a defined orientation, representing one-way relationships from one node to another. Additionally, graphs can be categorized by whether or not their edges carry weights.

An unweighted graph treats all edges equally without assigning any specific cost or value to them. Meanwhile, a weighted graph associates each edge with a numerical value, such as cost or distance.

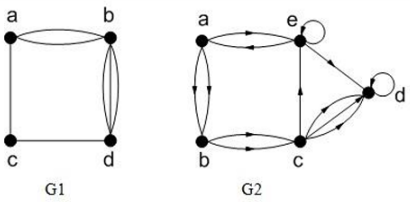

Fig. 2: Undirected and Directed Graph
source: https://informatika.stei.itb.ac.id/ rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf

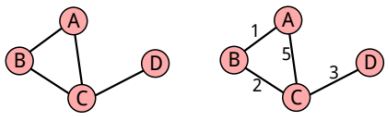

Fig. 3: Unweighted and Weighted Graph
source: https://osn.toki.id/data/pemrograman-kompetitif-dasar.pdf

There also exist graphs that combine characteristics from multiple types discussed above. For example, a graph can be both directed and weighted (Fig. 4).

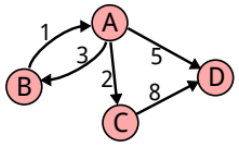

Fig. 4: Directed Weighted Graph
source: https://osn.toki.id/data/pemrograman-kompetitif-dasar.pdf

3) Subgraph
A subgraph is a graph formed from a subset of the vertices and edges of a larger graph. Formally, given a graph $G = (V,E)$, a graph $G_1 = (V_1,E_1)$ is called a subgraph of G if $V_1 \subseteq V$ and $E_1 \subseteq E$.

B. Osu!

Osu! is a free-to-play rhythm game in which players interact with visual hit objects that appear in synchronization with background music. Although the game includes multiple gameplay modes, such as *osu!taiko*, *osu!catch*, and *osu!mania*, this paper focuses exclusively on *osu!standard*, the default and most widely played mode. In *osu!standard*, players are required to click, slide, or spin hit objects at the correct time and in the correct sequence, based on both visual cues and musical rhythm.

Each beatmap in *osu!standard* represents a sequence of hit objects, which includes hit circles, sliders, and spinners. These objects are placed at specific coordinates on a 2D playfield and are accompanied by precise timing ring. The goal of the game is to interact with these objects as accurately as possible, in accordance with the music's tempo and structure. This makes the game both mechanically demanding and cognitively engaging, as it emphasizes timing precision, spatial awareness, and fast decision-making.

Fig. 5: osu! homescreen
source: Author's archive

Gameplay basics of osu!:
a) Playfield

Fig. 6: Playfield
source: https://osu.ppy.sh/wiki/en/Game_mode/osu!

b) Hit Circles
Hit Circles are simple objects that require the player to click on them precisely when a shrinking approach ring aligns with their border. They represent single-tap actions and are the most basic form of interaction.

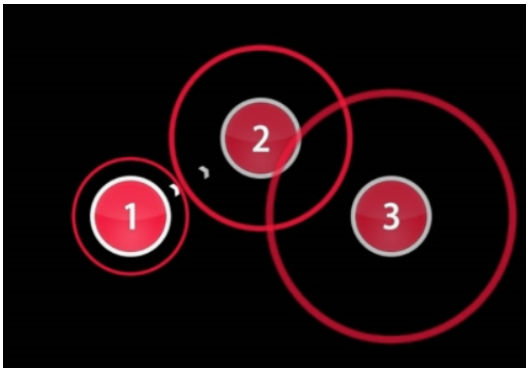

Fig. 7: Hit Circles
source: https://osu.ppy.sh/wiki/en/Game_mode/osu!

c) Sliders
Sliders consist of a start circle connected to a path that

the player must follow by holding the cursor along its trajectory until the end. Sliders often require players to track curved or zigzagging movements in rhythm with the music.

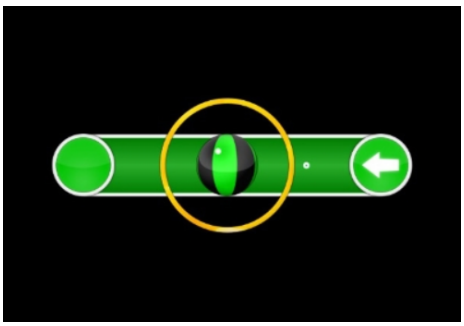

Fig. 8: Sliders

source: https://osu.ppy.sh/wiki/en/Game_mode/osu!

d) Spinners
Spinners require the player to rapidly spin their cursor around a central point for a set duration.

Fig. 9: Spinners

source: https://osu.ppy.sh/wiki/en/Game_mode/osu!

Osu! assigns each beatmap a star rating to indicate its overall difficulty, this value is the result of a complex simulation and is not directly interpretable by players. Osu! exposes several individual difficulty parameters, including:

- Approach Rate (AR): Determines how early hit objects appear before they must be interacted with.
- Overall Difficulty (OD): Controls the timing precision required to achieve higher accuracy scores.
- Circle Size (CS): Affects the size of hit objects, making them easier or harder to aim at.
- HP Drain Rate (HP): Influences how quickly a player's health depletes from misses and how much it regenerates from successful hits.

These values are shown explicitly to players and contribute significantly to how a beatmap "feels" while playing.

C. Beatmap Format and Object Properties

1) `.osu` File
`.osu` file is a human-readable file format containing information about a beatmap. Each `.osu` file is divided into several sections as follows, each sections indicated by section titles in square brackets.

TABLE I: `.osu` file structure [3]

Section	Description
`[General]`	General information about the beatmap
`[Editor]`	Saved settings for the beatmap editor
`[Metadata]`	Information used to indetify the beatmap
`[Difficulty]`	Difficulty settings
`[Events]`	Beatmap and storyboard graphic events
`[TimingPoints]`	Timing and control points
`[Colours]`	Combo and skin colours
`[HitObjects]`	Hit objects

This paper focuses specifically on the `[HitObjects]` section because it defines the individual interactive elements that appear during gameplay. By extracting and use the data from this section, it is possible to model the beatmap as a directed weighted graph. The structure of each line in the `[HitObjects]` sections follows a syntax: `x, y, time, type, hitSound, objectParams, hitSample` [3]
where:

- `x` (integer) and `y` (integer): Position in osu! pixels of the object.
- `time` (integer): Time when the object is to be hit, in milliseconds from the beginning of the beatmap's audio.
- `type` (integer): Bit flags indicating the type of the object.
- `hitSound` (integer): Bit flags indicating the hitsound applied to the object. See the hitsound section.
- `objectParams` (Comma-separated list): Extra parameters specific to the object's type.
- `hitSample` (Colon-separated list): Information about which samples are played when the object is hit. It is closely related to `hitSound`. If it is not written, it defaults to `0:0:0:0:`.

2) Temporal and Spatial Properties of Hit Objects
Each hit object in osu! beatmaps has both temporal and spatial characteristics, which define its gameplay challenge. Temporal properties is where each object has a time where it indicates when the object is to be hit. Spatial properties is where objects are mapped onto a 512x384 pixels playfield with specific (x, y) coordinates. These properties are parsed directly from the `.osu` file's `[HitObjects]` section. Together, they form the basis for modeling beatmaps as graphs.

III. Method

A. Limitations

The author decided to make a limit on the scope of this experiment to ensure a more focused and manageable analysis. The following boundaries are made:

- Game Mode: This paper only focuses on *osu!standard* game mode.

- Beatmap Section: The analysis is limited to the `[HitObjects]` section of the `.osu` file. Other sections are not included in the modeling.
- Hit Object Types: Among the hit objects, only *Hit Circles* and *Sliders* are considered. *Spinners* are excluded because they do not involve spatial movement between objects.
- Game Modifier: Modifications to beatmaps, such as *Hard Rock*, *Double Time*, *Hidden*, and *Easy* are excluded. Beatmaps are analyzed in their *No Mode* state.
- Difficulty Parameters: Approach Rate (AR), Overall Difficulty (OD), and Circle Size (CS) are not contributed into the graph weighting formula.
- Player Behaviour: Player's skill level, cursor movements, or reaction times are not considered. The focus is on the structural properties of the beatmap rather than player's individual skill.

B. Parsing osu! Beatmaps

The first step to model an osu! beatmap is to parse its `.osu` file to get four main information (`x`, `y`, `time`, and `type`) from the `[HitObjects]` section. The code shown in Fig. 10 parses the `[HitObjects]` section and extract important values that will be use to calculate edge weight. These values are stored as dictionaries in a list, which serves as the input for the graph construction. The parser ignores irrelevant sections and the spinners object types.

```python
# parse hit objects
def parse_hit_objects(osu_file_path):
    hit_objects = []
    parsing = False
    with open(osu_file_path, "r", encoding="utf-8") as f:
        for line in f:
            line = line.strip()
            if line.startswith("[HitObjects]"):
                parsing = True
                continue
            if parsing:
                if line == "" or line.startswith("["):
                    break
                parts = line.split(",")
                if len(parts) >= 5:
                    x = int(parts[0])
                    y = int(parts[1])
                    time = int(parts[2])
                    type_flag = int(parts[3])

                    if type_flag & 1:
                        obj_type = "circle"
                    elif type_flag & 2:
                        obj_type = "slider"
                    elif type_flag & 8:
                        obj_type = "spinner"
                    else:
                        obj_type = "other"

                    hit_objects.append({
                        "x": x,
                        "y": y,
                        "time": time,
                        "type": obj_type
                    })
    return hit_objects
```

Fig. 10: Parser function

source: Author's archive

C. Building the Graph

After parsing the valuable information from the beatmap, we can now build the graph where each node represents a hit object and each edge represents the transition to the next node. The edges are assigned with weights based on a custom difficulty formula. This implementation uses Python along with the `networkx` library to represent the beatmap graph and `matplotlib` library to visualize the resulting structure.

The code shown in Fig. 11 demonstrates how the graph is built using the previously parsed hit objects using `networkx` library. Each hit object is added as a node with its attribute, such as position, time, and hit object type. Directed edges are then created between consecutive nodes.

The code shown in Fig. 12 visualizes the constructed graph using the `matplotlib` library. Nodes are positioned based on their in-game position. Different colors are used to distinguish between object types (blue for *circles*, green for *sliders*). Note that the nodes in the graph are 0-index.

```python
# construct the graph
def build_graph(hit_objects):
    G = nx.DiGraph()
    for i, obj in enumerate(hit_objects):
        G.add_node(i, **obj)
    for i in range(len(hit_objects) - 1):
        obj1 = hit_objects[i]
        obj2 = hit_objects[i + 1]
        w = calculate_difficulty(obj1, obj2)
        print(f"Edge from {i} to {i+1}, weight = {w:.2f}")
        G.add_edge(i, i + 1, weight=w)
    return G
```

Fig. 11: Build graph function

source: Author's archive

```python
# visualization
def visualize_graph(G):
    pos = {node: (G.nodes[node]['x'], -G.nodes[node]['y']) for node in G.nodes}

    node_colors = []
    for n in G.nodes:
        obj_type = G.nodes[n]['type']
        if obj_type == "circle":
            node_colors.append("blue")
        elif obj_type == "slider":
            node_colors.append("green")
        else:
            node_colors.append("gray")

    plt.figure(figsize=(10, 6))
    nx.draw(G, pos, node_color=node_colors, with_labels=True, node_size=300,
            font_size=8, edge_color='gray', arrows=True)

    # edge weight
    for u, v, data in G.edges(data=True):
        x1, y1 = pos[u]
        x2, y2 = pos[v]
        label_x = (x1 + x2) / 2
        label_y = (y1 + y2) / 2
        weight = data['weight']
        plt.text(label_x, label_y, f"{weight:.1f}", fontsize=6, color='red', ha='center', va='center')

    plt.show()
```

Fig. 12: Graph visualization function

source: Author's archive

D. Custom Edge Weight Formula

The author used a custom formula to calculate each edge weight. The weight is influenced by two main factors:

- Spatial Distance: It's the Euclidean distance between two nodes. Larger distances indicate harder transition.
- Temporal Interval: The time difference between two nodes. Smaller intervals require quicker reactions, increasing difficulty.

In addition, the author also consider two special sequences in a beatmap, a burst and a stream. Burst is where many nodes are in the exact same position, only difference is in time interval. These are often part of rapid sequences that are mechanically demanding, even if the player's cursor doesn't need to move. Burst difficulty is addressed by assigning a bonus to consecutive objects at the same position. Stream is a pattern of multiple hit circles placed in rapid succession, often at a consistent distance and timing interval and they significantly increase the difficulty of the beatmap. The author's custom formula for edge weight w of two consecutive nodes is defined as:

$$w = \begin{cases} \frac{B}{\Delta t} \cdot \gamma \cdot \alpha, & \text{if } d = 0 \quad \text{(burst bonus)} \\ \frac{d}{\Delta t} \cdot \gamma \cdot \alpha, & \text{otherwise} \end{cases} \tag{1}$$

where:

- d is the Euclidean distance between two nodes.
- Δt is the time difference between two nodes.
- B is a fixed burst bonus value (the author used 100 for experiment).
- γ is a scale factor (the author uses 5 for experiment).
- α is a stream factor.

The code shown in Fig. 13 implements the custom edge weight formula used to estimate the difficulty between two consecutive nodes. When two nodes appear in the exact same position within a short time frame, the function applies a *burst bonus* to increase the edge weight. The result is then multiplied by stream factor, if the temporal interval is less than 100 milliseconds and the spatial distance between current edge and the previous edge is less than 1.5, stream is detected and the stream factor is set to be 6, otherwise it follows the default value, which is 1. This adjustment ensures that high-speed sequences are not underrated.

```python
# weight formula
def calculate_difficulty(obj0, obj1, obj2, burst_bonus=100):
    dx = obj2["x"] - obj1["x"]
    dy = obj2["y"] - obj1["y"]
    spatial_dist = math.sqrt(dx**2 + dy**2)

    prev_dx = obj1["x"] - obj0["x"]
    prev_dy = obj1["y"] - obj0["y"]
    prev_dist = math.sqrt(prev_dx**2 + prev_dy**2)
    delta_spatial = abs(spatial_dist - prev_dist)

    if delta_time == 0:
        delta_time = 1

    scale_factor = 5
    stream_factor = 1
    if (delta_time <= 100 and delta_spatial <= 1.5):
        stream_factor = 3
        scale_factor = 6
    if spatial_dist == 0:
        return (burst_bonus / delta_time)*(scale_factor*stream_factor)
    else:
        return (spatial_dist / delta_time)*(scale_factor*stream_factor)
```

Fig. 13: Custom weight formula function

source: Author's archive

E. Difficulty Estimation and Execution

The final step is to run the program using a driver that executes the difficulty estimation process (Fig. 14). The driver program begins by parsing the beatmap file, build a directed graph, assigning edge weights, and visualizing the resulting graph structure.

Once the graph is built, the program proceeds to estimate the difficulty of the beatmap by calculating the average edge weight across the entire graph. This average weight reflects overall gameplay intensity and difficulty.

The author chooses to classify beatmaps into intuitive difficulty estimation based on the average weight of the graph. These ranges are as follows:

- Easy (< 3): Represents beatmaps with slow tempo or widely spaced objects.
- Challenging $(3-5)$: Represents beatmaps with moderate tight timing or spatial transitions.
- Medium-Well $(5-6.5)$: Represents beatmaps with more tight timing or spatial transitions with consistent bursts or directional changes.
- Hard (> 6.5): Represents beatmaps with high-density patterns, such as jumps, bursts, and streams.

Although informal, these classification gives players a quick, interpretable estimate of a beatmap's overall intensity and difficulty.

```python
# driver
if __name__ == "__main__":
    osu_file = "beatmaps/TRUE - Soundscape (SkyFlame) [Kowari's Expert].osu"
    hit_objects = parse_hit_objects(osu_file)
    print(f"Parsed {len(hit_objects)} hit objects.")

    G = build_graph(hit_objects)
    print(f"Graph: {len(G.nodes)} nodes, {len(G.edges)} edges.")

    average = average_weight(G)
    print(f"Average weight: {average:.2f}")

    if (average < 3):
        print("This map is Easyyyy!")
    elif (average >= 3 and average < 5):
        print("This map is quite challenging")
    elif (average >= 5 and average < 6.5):
        print("This map is medium well")
    else:
        print("This map is hard, you are cooked bro")

    visualize_graph(G)
```

Fig. 14: Driver program

source: Author's archive

The full implementation code used in this experiment can be reviewed in the appendix.

IV. Experiment

The author chooses a beatmap sets titled *"TRUE - Soundscape (SkyFlame)"* from the osu! beatmaps repository [5]. This set contains eight distinct difficulty levels, providing a wide range of gameplay complexity. For the purpose of this experiment, the author selects three out of eight maps, which are the *"[Easy]"*, *"[Kowari's Expert]"*, and *"[Melody]"*. This selection allows the graph-based difficulty estimation method to be evaluated across varying beatmap difficulties.

1) *TRUE - Soundscape (SkyFlame) [Easy]*
   According to the beatmap set, this beatmap has an official star rating at 2.06. Its complete graph representation is shown in Fig. 15. As printed in the terminal output (Fig. 16), the graph contains 428 nodes and 427 edges. A few sample edges and their computed weights are also printed. This graph's average edge weight is 1.35.

Based on the classification the author has defined, this beatmap is categorized as “Easy” in difficulty, indicating a relatively low gameplay difficulty.

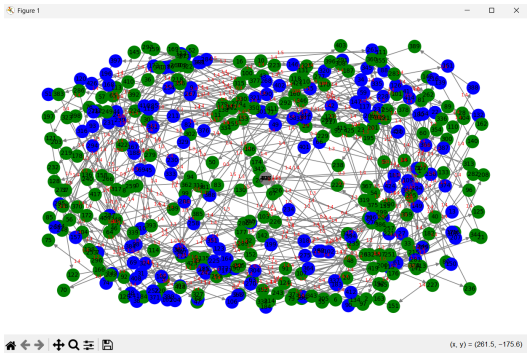

Fig. 15: *[Easy]*’s full graph
source: Author’s archive

```
Edge from 416 to 417, weight = 1.40
Edge from 417 to 418, weight = 1.26
Edge from 418 to 419, weight = 1.42
Edge from 419 to 420, weight = 1.47
Edge from 420 to 421, weight = 1.40
Edge from 421 to 422, weight = 1.38
Edge from 422 to 423, weight = 1.41
Edge from 423 to 424, weight = 1.32
Edge from 424 to 425, weight = 1.41
Edge from 425 to 426, weight = 1.37
Edge from 426 to 427, weight = 3.77
Graph: 428 nodes, 427 edges.
Average weight: 1.35
This map is Easyyyy!
```

Fig. 16: *[Easy]*’s terminal output
source: Author’s archive

2) *TRUE - Soundscape (SkyFlame) [Kowari’s Expert]*
According to the beatmap set, this beatmap has an official star rating at 5.85. Its complete graph representation is shown in Fig. 17. A subgraph is shown in Fig. 18 to provide a clearer view of individual nodes, edges, and their corresponding weights. As printed in the terminal output (Fig. 19), this beatmaps has 1311 nodes, 1310 edges, and the average weight is 6.32. Based on the classification the author has defined, this beatmap is categorized as “Medium-Well” in difficulty, indicating more spatial transitions with consistent bursts or directional changes. This beatmap is harder than the previous one (*TRUE - Soundscape (SkyFlame) [Easy]*).

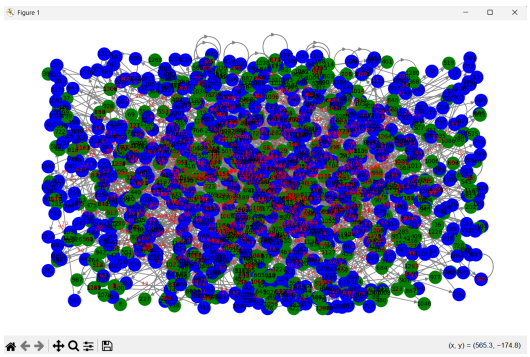

Fig. 17: *[Kowari’s Expert]*’s full graph
source: Author’s archive

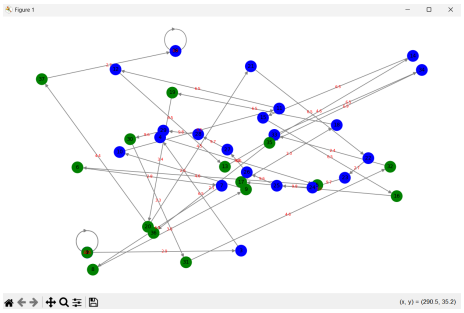

Fig. 18: *[Kowari’s Expert]*’s subgraph
source: Author’s archive

```
Edge from 1299 to 1300, weight = 7.37
Edge from 1300 to 1301, weight = 7.50
Edge from 1301 to 1302, weight = 6.82
Edge from 1302 to 1303, weight = 6.86
Edge from 1303 to 1304, weight = 6.83
Edge from 1304 to 1305, weight = 6.84
Edge from 1305 to 1306, weight = 6.84
Edge from 1306 to 1307, weight = 6.17
Edge from 1307 to 1308, weight = 22.22
Edge from 1308 to 1309, weight = 6.70
Edge from 1309 to 1310, weight = 5.76
Graph: 1311 nodes, 1310 edges.
Average weight: 6.32
This map is medium well
```

Fig. 19: *[Kowari’s Expert]*’s terminal output
source: Author’s archive

3) *TRUE - Soundscape (SkyFlame) [Melody]*
According to the beatmap set, this beatmap has an official star rating at 6.8. Its complete graph representation is shown in Fig. 20. As printed in the terminal output (Fig. 21), the graph contains 1346 nodes and 1345 edges. This graph’s average weight is 7.12, which based on the classification the author has defined, this beatmap is categorized as “Hard” in difficulty, indicating high-density patterns. This beatmap is also the hardest of the beatmap set according to the osu! beatmap repository.

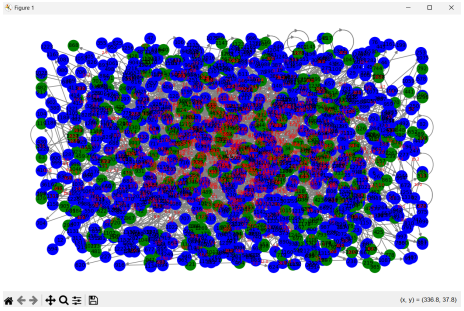

Fig. 20: *[Melody]*’s full graph
source: Author’s archive

```
Edge from 1335 to 1336, weight = 11.59
Edge from 1336 to 1337, weight = 12.90
Edge from 1337 to 1338, weight = 8.16
Edge from 1338 to 1339, weight = 2.66
Edge from 1339 to 1340, weight = 9.43
Edge from 1340 to 1341, weight = 2.75
Edge from 1341 to 1342, weight = 2.56
Edge from 1342 to 1343, weight = 2.71
Edge from 1343 to 1344, weight = 9.67
Edge from 1344 to 1345, weight = 6.17
Graph: 1346 nodes, 1345 edges.
Average weight: 7.12
This map is hard, you are cooked bro
```

Fig. 21: *[Melody]*’s output terminal
source: Author’s archive

V. Conclusion

In this paper, we successfully modeled osu! beatmaps into graph to estimate its gameplay difficulty in a more interpretable and structured way. By parsing `.osu` files, the sequences of hit objects can be represented as a directed graph, where each node represents a hit object and each edge represents the transition between them. A custom edge weight formula was also defined to assign weight to each edge by taking into account of spatial distance, time interval, and specific gameplay patterns, such as bursts and streams.

Through this graph-based modeling, the average edge weight of all the existing edges was used as an estimation for gameplay difficulty, and an intuitive classification system was defined to label beatmaps as Easy, Challenging, Medium-Well, or Hard. After experimenting with several selected beatmaps, results show that the method used was quite aligned with the osu!'s official star rating system, indicating that this approach captures meaningful difficulty characteristics of the game.

However, while the graph-based model provides a structured approach to estimate a beatmap difficulty, there remains room for improvement. The current method does not yet account for more nuanced gameplay elements, such as cursor movement complexity, sliders pattern, and game modifier. These included aspects would allow for a more comprehensive and accurate estimation that better reflects the wide variety of skillsets required in osu! gameplay.

VI. Appendix

- Source code: *https://github.com/Valz0504/Modeling-osu-beatmaps.git*
- Video: *https://youtu.be/JfTSoxNzPiU*

VII. Acknowledgement

The author would like to express sincere gratitude to Dr. Ir. Rinaldi, M.T. and Mr. Arrival Dwi Sentosa, S.Kom., M.T., lecturers of the IF1220 Discrete Mathematics course, for their guidance, encouragement, and support throughout the semester. Thanks to their support, the author was able to gain valuable insights and learn many new things throughout the course. The author is also deeply thankful to family and friends who have been keep motivating and supporting.

References

[1] R. Munir, "Graf (Bagian 1)", [Online]. Available: `https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf`. [Accessed: 17-Jun-2025].

[2] ppy, "osu!standard," *osu! wiki*, [Online]. Available: `https://osu.ppy.sh/wiki/en/Game_mode/osu!`. [Accessed: 17-Jun-2025].

[3] ppy, "`.osu` (file format)," *osu! wiki*, [Online]. Available: `https://osu.ppy.sh/wiki/en/Client/File_formats/osu_(file_format)`. [Accessed: 17-Jun-2025].

[4] GeeksforGeeks, "Introduction to Graphs in Python," [Online]. Available: `https://www.geeksforgeeks.org/python/introduction-to-graphs-in-python/`. [Accessed: 18-Jun-2025].

[5] osu!, "osu! Beatmap Listing," [Online]. Available: `https://osu.ppy.sh/beatmapsets`. [Accessed: 18-Jun-2025].

Statement

Hereby, I declare that this paper I have written is my own work, not a reproduction or translation of someone else's paper, and not plagiarized.

Bandung, 19 January 2025

Emilio Justin (13524043)