

Tonnetz as a Planar Graph Representation for Generating Harmonic Chord Progressions in Songs

Alya Nur Rahmah - 13524081

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung, Jalan Ganesha 10 Bandung

E-mail: alyanrrmah@gmail.com , 13524081@std.stei.itb.ac.id

Abstract— Harmonic structures in music can be analyzed and shaped through systematic mathematical approaches, one of which is graph representation. Tonnetz, a conceptual lattice diagram representing tonal space that describes the relationship between tones based on harmonic intervals such as perfect fifth and major third, can be modeled as a planar graph to help the process of automatic generation of chord progressions. In this paper, Tonnetz is modeled as a graph, where each vertex represents a major or minor chord, and edges connect pairs of chords that have harmonic proximity. By applying graph theory concepts from discrete mathematics, graph traversal simulations are performed that generate a sequence of chord progressions with high harmonic connectivity. Graph visualization shows that the Tonnetz structure enables the construction of coherent progression paths and can be used in digital music creation. Experimental results show that this method can be used as a logical approach to support automatic chord composition or improvisation systems.

Keywords— *Tonnetz, graph theory, planar graph, chord progressions.*

I. INTRODUCTION

Music is one of the forms of artistic expression that has developed rapidly alongside advances in technology and science. One of the fundamental aspects of music is harmony, which refers to the orderly relationship between notes that form chord progressions and give a composition its distinctive character. Harmonious chord progressions are key to creating the mood, emotion, and musical appeal of a song. Therefore, the analysis and design of good chord progressions are important considerations in music theory and the development of digital music applications.

In recent decades, mathematical approaches have been widely used to understand and develop musical structures. One of the approach is the representation of note and chord relationships in the form of graphs. Graph theory, as part of discrete mathematics, offers a formal framework that can be used to systematically model and analyze complex relationships between musical elements. By representing notes or chords as vertices and harmonic relationships as edges, various patterns and progression paths can be explored.

One of the most famous graph models in music theory is the Tonnetz, first introduced by Leonhard Euler in the 18th century. The Tonnetz is a lattice diagram that depicts

the tonal space based on harmonic intervals such as the perfect fifth and major third. In Tonnetz, notes or chords that are harmonically close are geometrically close, making it easier to analyze and create coherent chord progressions. In this model, each vertex represents a major or minor chord, while edges connect chords with harmonic proximity. By modeling Tonnetz as a planar graph, the process of finding chord progression paths can be done through graph traversal algorithms, enabling the automatic generation of chord progressions with high harmonic connectivity. . By using graph traversal algorithms, chord progression paths can be systematically generated while considering strong harmonic relationships between chords. This approach provides a logical and structured method to support the automatic composition of chords.

The use of Tonnetz as a graph model for automatic chord progression generation is highly relevant in the context of developing digital music applications, such as automatic composition systems and improvisation tools. This approach not only strengthens the connection between music theory and discrete mathematics but also opens opportunities for the development of more advanced and adaptive music technology. Through graph traversal simulation on Tonnetz, coherent and harmonious chord progressions can be generated efficiently, thereby supporting creativity and innovation in music composition.

This paper discusses the modeling of Tonnetz as a planar graph connecting major and minor chords based on harmonic intervals, as well as the implementation of graph traversal simulation to generate automatic chord progressions. This research aims to model Tonnetz as a graph, implement graph traversal simulations to generate automatic chord progressions, and analyze the results both visually and musically. The simulation results will be visually analyzed to assess the harmonic relationships and coherence of the generated chord progressions. Thus, this paper is expected to contribute to the development of applicable mathematical methods to support the automatic and systematic creation of digital music.

II. BASIC THEORY

A. Graph

1) Definition

A graph is a structure used to represent discrete objects and the relationships between them. Formally, a graph is defined as an ordered pair represented as $G = (V, E)$, where V is a non-empty set of vertex (v_1, v_2, \dots, v_n) and E is a set of edges (e_1, e_2, \dots, e_n) connecting two vertex in V .

There are several variations of graph types based on the presence of multiple edges and loop edges. If a graph has no multiple edges or edges that return to the original vertex (loop edges), it is called a simple graph. A graph containing multiple edges is called a multi-graph, and a graph with edges connecting a vertex to itself (loop edges) is called a pseudograph.

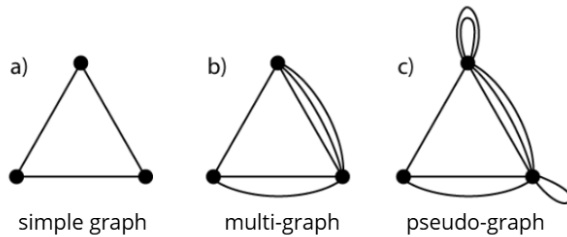


Figure 1.1.1 Graph types based on multiple and loop edges (source : [1])

Therefore, graphs can also be classified based on the orientation of their edges. A graph that does not have a direction on each edge is called an undirected graph, while a graph that has a direction is called a directed graph or digraph. In a directed graph, each edge is represented as an ordered pair indicating the direction from one vertex to another.

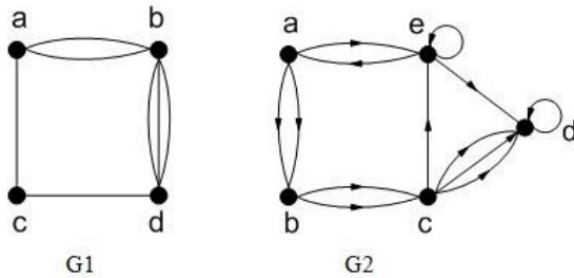


Figure 1.1.2 (G1) undirected graph (G2) directed graph (source : [1])

In addition, there are several special graphs below

- Complete Graph** : A graph in which each vertex is directly connected to all other vertex. A complete graph with vertex n is denoted as K_n , and has $n(n-1)/2$ edges.
- Cycle Graph** : A simple graph in which each vertex has degree two and forms a closed cycle.
- Regular Graph** : A graph in which every vertex has the same degree. If the degree is r , and the number of vertices is n , then the number of edges is $nr/2$.

- Bipartite Graph** : A graph whose vertices can be divided into two sets V_1 and V_2 , such that each edge connects only a vertex from V_1 to a vertex in V_2 . This graph is denoted as $G = (V_1, V_2)$.

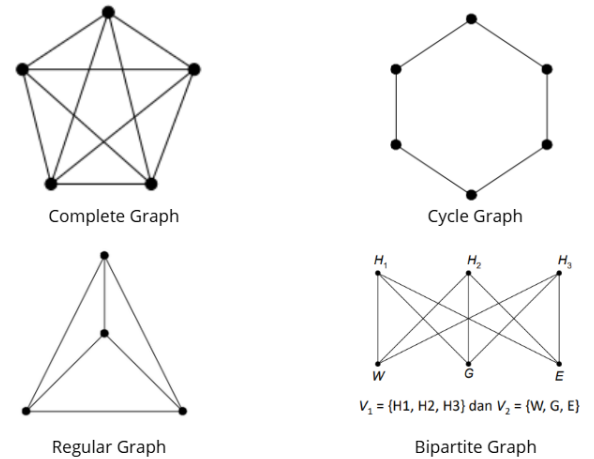


Figure 1.1.3 Another special graph (source : [1])

2) Terminology

- Adjacency** : Two vertex are said to be adjacent if they are directly connected by an edge.
- Incidence**: An edge is said to be incident to two vertex if it connects those two vertices.
- Isolated Vertex**: A vertex that is not connected to any other vertex or has no edges at all.
- Null Graph or Empty Graph**: A graph consisting of a number of vertices but no edges.
- Degree** : The number of edges adjacent to a vertex. In a directed graph, there is in-degree (number of incoming edges) and out-degree (number of outgoing edges).
- Path** : A sequence of vertices connected by edges in order without repeating edges.
- Cycle or Circuit** : A closed path that starts and ends at the same vertex without repeating any edges.
- Connected** : Two vertices are said to be connected if there is a path between them. A graph is said to be connected if all pairs of vertices are connected.
- Subgraph and Complement Subgraph** : A subgraph is a part of a graph consisting of some of the vertices and edges of the original graph. The complement subgraph is a graph with the same set of vertices, but its edges are the complement of the edges of the original graph.
- Spanning Subgraph** : A subgraph that contains all the vertices of the original graph and is part of that graph.
- Cut Set** : A set of edges that, if removed, would cause the graph to become disconnected.
- Weighted Graph** : A graph in which each edge has a specific value or weight, representing a particular thing.

3) Representation

a) **Adjacency Matrix** : A matrix that represents the relationships between nodes. The value in row i and column j is 1 if nodes i and j are connected. In a directed graph, the direction is indicated by the location of the value 1. If the graph is weighted, the value in that position indicates the weight of the edge.

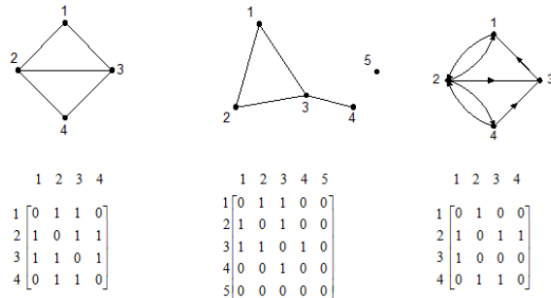


Figure 1.3.1 Adjacency matrix (source : [2])

b) **Incidence Matrix**: A matrix that shows the relationship between nodes and edges. The elements of the matrix are 1 if the i th node is connected to the j th edge, and 0 if not. For directed graphs, the values can be -1 and 1 depending on the direction of the edge.

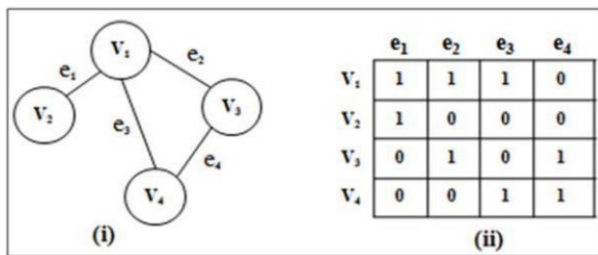


Figure 1.3.2 Incidence matrix (source : [2])

c) **Adjacency List** : Each node is wrote along with a list of its neighboring nodes. This representation is efficient for sparse graphs.

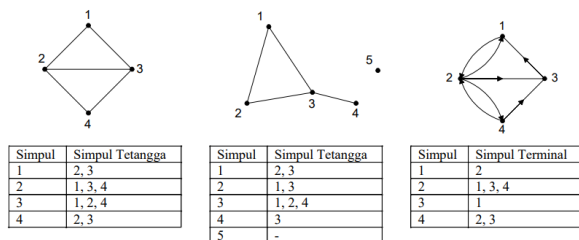


Figure 1.3.3 Adjacency list (source : [2])

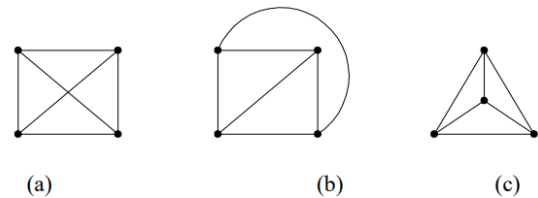


Figure 2.1 (a) planar graph (b) and (c) plane graph (source : [2])

One of the fundamental concepts in planar graphs is Euler's Formula, which states the following:

$$n - e + f = 2$$

where:

- n : number of vertex,
- e : number of edges,
- f : number of regions (faces) including the outer region.

This equation applies to simple connected planar graphs. This Euler characteristic forms the basis for analyzing the structure of planar graphs and can determining the possibility of planarity of a graph based on the number of vertex and edges.

In addition to the Euler equation, there is the Euler inequality for simple planar graphs with $e > 2$, where:

$$e \leq 3n - 6$$

If a graph does not satisfy this inequality, then it is not planar. Therefore, if it satisfies the inequality, it is not necessarily planar, but it could be a candidate for a planar graph.

The planarity of a graph can also be tested using Kuratowski's theorem, which states that a graph G is planar if and only if G does not have a subgraph that is isomorphic or homeomorphic to one of the two classical non-planar graphs, $K_{3,3}$ and K_5 . These two graphs are called Kuratowski graphs. This theorem is very useful for proving that a graph is not planar by showing the existence of a subgraph identical to a Kuratowski structure.

C. Chord Progression

Chord progression is a sequence of chord changes played sequentially in a musical composition. This structure forms the harmonic framework of a song and greatly influences the mood, emotion, and musical direction of the piece. In music theory, common chord progressions often follow certain patterns such as I-IV-V-I or II-V-I, which are considered harmonically stable and pleasing to the listener.

Each chord in the progression has a specific function; the tonic as the center of stability, the dominant creating tension, and the subdominant serving as a bridge between the two. These functions form the foundation of tonal harmony theory. Additionally, the selection and sequencing of chords can convey specific nuances, ranging from cheerful to melancholic, from tense to a calm resolution.

In the context of music technology and mathematics, chord progressions can also be represented graphically, where each chord becomes a node and the transitions between chords become edges. This approach enables the application of graph algorithms to explore, analyze, and even automatically generate new chord progressions.

B. Planar Graph

A planar graph is a graph that can be drawn on a flat surface without any edges crossing each other. If a planar graph is drawn in such a way that no edges cross each other, then the resulting drawing is called a plane graph. Planar graphs are often used in various practical applications, such as electronic circuit design, where the depiction of non-overlapping paths is essential to prevent electrical interference.

D. Tonnetz

Tonnetz, is a geometric representation that illustrates the harmonic relationships between tones in tonal music, particularly those related to the major third (M3), minor third (m3), and perfect fifth (P5) intervals. This concept was first introduced by Leonhard Euler in the 18th century and later further developed by modern music theorists within the context of Neo-Riemannian theory.

In a two-dimensional Tonnetz representation, each node represents a pitch class, and the edges connect notes with strong harmonic relationships, such as forming a major or minor triad. Thus, each triangle in the Tonnetz represents a triad chord. When connected as a whole, the Tonnetz forms a triangular lattice where movement between chords can be visualized as transitions between nodes or paths on a planar graph.

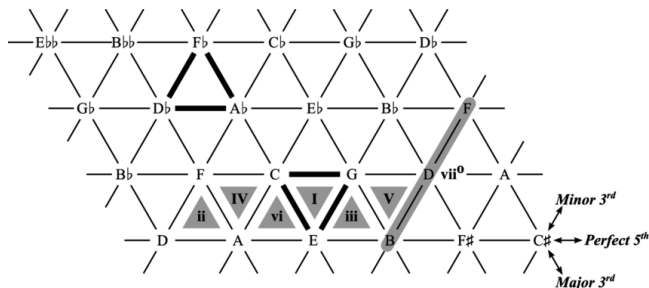


Figure 1.4 Tonnetz (source : [6])

The structure of the Tonnetz facilitates understanding of the harmonic proximity between chords. For example, two chords adjacent to each other in the Tonnetz have overlapping notes (minimal voice leading), making them suitable for creating chord progressions that sound natural and coherent. Therefore, many music computing systems utilize this structure to automatically generate or analyze chord progressions.

III. IMPLEMENTATION

A. Representing Tonnetz as Planar Graph

The Tonnetz structure is represented as an undirected graph, where the nodes represent pitch classes (notes such as C, D, E, and so on), and the edges represent harmonic relationships in the form of triads (groups of three notes).

To ensure that the graph formed is truly planar, the representation is limited to a local subset of the Tonnetz, consisting of several major and minor triads that are harmonically related to each other. These triads partially overlap, forming neatly arranged triangles that do not intersect, as shown in the geometric Tonnetz grid.

Examples of triads selected in this representation include:

- C major: (C, E, G)
- A minor: (A, C, E)
- F major: (F, A, C)
- D minor: (D, F, A)

Each triad is formed as a triangle, and its vertices are connected to form the sides between harmonically related

notes. The graph is then constructed using the `networkx` library and visualized using `matplotlib`.

```
import networkx as nx
import matplotlib.pyplot as plt

# Contoh subset triad lokal yang harmonik
triads = [
    ('C', 'E', 'G'),
    ('A', 'C', 'E'),
    ('F', 'A', 'C'),
    ('D', 'F', 'A')
]

G = nx.Graph()

for triad in triads:
    a, b, c = triad
    G.add_edge(a, b)
    G.add_edge(b, c)
    G.add_edge(c, a)

# Visualisasi graf
pos = nx.planar_layout(G)
nx.draw(G, pos, with_labels=True, node_color='lightblue', node_size=1000, font_size=12)
plt.title("Representasi Tonnetz sebagai Graf Planar")
plt.show()
```

Figure 3.1.1 Representing Tonnetz as Planar Graph using Python

B. Generating Harmonic Chord Progressions and Classification Based On Song Genre

The main implementation process in this research is to represent the Tonnetz structure as a planar graph, then utilize this structure to generate automatic chord progressions tailored to the characteristics of various music genres. The graph is constructed using the Python networkx library, while the probabilistic logic and chord transition processing are performed through the ChordNode representation.

Each node in the graph represents a chord (in the form of pitch class and quality such as major, minor, or diminished). The edges between nodes reflect harmonic relationships based on musical transformations such as parallel (P), relative (R), leading-tone (L), as well as fifth (V) and fourth (IV) movements.

Once the graph is formed, chord progressions are generated by traversing from the initial chord to its adjacency. The selection is not purely random but follows a weighted random selection scheme considering the harmonic weight of the edge, genre-specific weight, also enhancing genre influence by squaring the genre. With this approach, the resulting chord progression is not only musical but also reflects the distinctive nuances of the selected genre.

```
class ChordNode:
    def __init__(self, root, quality):
        self.root = root
        self.quality = quality # 'major', 'minor', 'dim', 'aug'

    def __str__(self):
        if self.quality == 'major':
            return self.root
        elif self.quality == 'minor':
            return f"#{self.root}m"
        elif self.quality == 'dim':
            return f"#{self.root}m"
        elif self.quality == 'aug':
            return f"#{self.root}+"
        return f"{self.root}{self.quality}"

    def __hash__(self):
        return hash((self.root, self.quality))

    def __eq__(self, other):
        if not isinstance(other, ChordNode):
            return False
        return self.root == other.root and self.quality == other.quality
```

Figure 3.2.1 Class to represent a chord

The image above represents the basic structure of a chord in this system. The chord object is created using the `__init__()` constructor with the root parameter as the base note (e.g., C,

D#, or F#) and the quality parameter as the chord type (major, minor, dim, or aug). The `__str__()` function is used to return a string representation of the chord to be displayed to the user, for example C for major, Am for minor, and F#° for diminished. The implementation of the `__hash__()` and `__eq__()` functions aims to enable chord objects to be used as nodes in a networkx graph, while ensuring object equality based on value rather than just identity.

```
class TonnnetzGraph:
    def __init__(self):
        self.graph = nx.Graph()
        self.notes = ['C', 'C#', 'D', 'D#', 'E', 'F', 'F#', 'G', 'G#', 'A', 'A#', 'B']
        self.chord_nodes = {}
        self.genre_weights = self._initialize_genre_weights()
        self._build_tonnetz()

    def _initialize_genre_weights(self):
        weights = {}

        # Jazz - lebih banyak diminished dan transisi kompleks
        weights['jazz'] = {
            ('major', 'minor'): 0.8, ('minor', 'major'): 0.7,
            ('major', 'dim'): 0.9, ('dim', 'major'): 0.8,
            ('minor', 'minor'): 0.6, ('major', 'major'): 0.4,
            ('dim', 'minor'): 0.7, ('minor', 'dim'): 0.8,
            ('major', 'aug'): 0.3, ('aug', 'minor'): 0.5
        }

        # Rock - dominasi major chord
        weights['rock'] = {
            ('major', 'major'): 0.9, ('minor', 'major'): 0.8,
            ('major', 'minor'): 0.7, ('minor', 'minor'): 0.6,
            ('major', 'dim'): 0.3, ('dim', 'major'): 0.4,
            ('dim', 'minor'): 0.2, ('minor', 'dim'): 0.2,
            ('major', 'aug'): 0.1, ('aug', 'minor'): 0.2
        }

        # Pop - balanced
        weights['pop'] = {
            ('major', 'major'): 0.8, ('major', 'minor'): 0.9,
            ('minor', 'major'): 0.8, ('minor', 'minor'): 0.5,
            ('major', 'dim'): 0.4, ('dim', 'major'): 0.5,
            ('dim', 'minor'): 0.3, ('minor', 'dim'): 0.3,
            ('major', 'aug'): 0.2, ('aug', 'minor'): 0.3
        }

        # Hip-hop - dominasi minor
        weights['hiphop'] = {
            ('minor', 'minor'): 0.9, ('minor', 'major'): 0.7,
            ('major', 'minor'): 0.9, ('major', 'major'): 0.5,
            ('minor', 'dim'): 0.4, ('dim', 'minor'): 0.6,
            ('major', 'dim'): 0.3, ('dim', 'major'): 0.3,
            ('major', 'aug'): 0.1, ('aug', 'minor'): 0.2
        }

        # Classical - harmoni kompleks
        weights['classical'] = {
            ('major', 'major'): 0.7, ('major', 'minor'): 0.8,
            ('minor', 'major'): 0.8, ('minor', 'minor'): 0.6,
            ('major', 'dim'): 0.9, ('dim', 'major'): 0.8,
            ('dim', 'minor'): 0.9, ('minor', 'dim'): 0.8,
            ('major', 'aug'): 0.4, ('aug', 'minor'): 0.5
        }

        return weights
```

Figure 3.2.2 Code of the TonnnetzGraph constructor and the `_initialize_genre_weights()` function

The `__init__()` constructor is used to initialize an empty graph, define a 12-tone equal temperament scale, form chord nodes with three types of qualities (major, minor, diminished), and set transition weights between chords based on genre through the `_initialize_genre_weights()` function.

```
class TonnnetzGraph:
    def _build_tonnetz(self):
        for note in self.notes:
            for quality in ['major', 'minor', 'dim']:
                chord = ChordNode(note, quality)
                self.chord_nodes[str(chord)] = chord
                self.graph.add_node(chord)
            self._add_tonnetz_edges()

    def _add_tonnetz_edges(self):
        for chord in self.chord_nodes.values():
            if chord.quality == 'major':
                parallel_chord = ChordNode(chord.root, 'minor')
                if parallel_chord in self.graph.nodes:
                    self.graph.add_edge(chord, parallel_chord, transform='P', weight=1.0)
            if chord.quality == 'minor':
                rel_root = self._get_relative_minor(chord.root)
                rel_chord = ChordNode(rel_root, 'minor')
                if rel_chord in self.graph.nodes:
                    self.graph.add_edge(chord, rel_chord, transform='R', weight=0.9)
            if chord.quality == 'dim':
                lead_root = self._get_leading_tone_major(chord.root)
                lead_chord = ChordNode(lead_root, 'major')
                if lead_chord in self.graph.nodes:
                    self.graph.add_edge(chord, lead_chord, transform='L', weight=0.8)
                fifth_root = self._get_fifth(chord.root)
                fifth_chord = ChordNode(fifth_root, chord.quality)
                if fifth_chord in self.graph.nodes:
                    self.graph.add_edge(chord, fifth_chord, transform='V', weight=0.9)
                fourth_root = self._get_fourth(chord.root)
                fourth_chord = ChordNode(fourth_root, chord.quality)
                if fourth_chord in self.graph.nodes:
                    self.graph.add_edge(chord, fourth_chord, transform='IV', weight=0.7)
```

Figure 3.2.3 Code of the `_build_tonnetz()` function and the `_add_tonnetz_edges()` function

The `_build_tonnetz()` function is responsible for constructing all nodes in the Tonnnetz graph and storing them in the networkx graph structure. After all chord nodes have been added, the `_add_tonnetz_edges()` function will be called to form edges between nodes. The relationships between these chords are based on the principles of Neo-Riemannian harmonic transformation, such as P (Parallel), R (Relative), L (Leading-tone), V (Dominant/Fifth), and IV (Subdominant/Fourth).

```
class TonnnetzGraph:
    def _get_relative_minor(self, major_root):
        idx = self.notes.index(major_root)
        return self.notes[(idx - 3) % 12]

    def _get_leading_tone_major(self, minor_root):
        idx = self.notes.index(minor_root)
        return self.notes[(idx + 3) % 12]

    def _get_fifth(self, root):
        idx = self.notes.index(root)
        return self.notes[(idx + 7) % 12]

    def _get_fourth(self, root):
        idx = self.notes.index(root)
        return self.notes[(idx + 5) % 12]

    def _parse_chord(self, chord_str):
        chord_str = chord_str.strip()

        if chord_str.endswith(':'):
            return ChordNode(chord_str[:-1], 'dim')
        elif chord_str.endswith('#'):
            return ChordNode(chord_str[:-1], 'aug')
        elif chord_str.endswith('m'):
            return ChordNode(chord_str[:-1], 'minor')
        else:
            return ChordNode(chord_str, 'major')
```

Figure 3.2.4 Helper function code in class TonnnetzGraph

The code above displays helper functions such as `_get relative_minor()`, `_get_leading_tone_major()`, `_get_fifth()`, and `_get_fourth()`. Each function is responsible for calculating tone transformations according to specific intervals used to construct the inner side of the graph. The `_parse_chord()` function is responsible for converting user input strings (e.g., Am, C#°) into ChordNode objects.

```
class TonnnetzGraph:
    def generate_progression(self, start_chord, length, genre='pop'):
        try:
            current_chord = self._parse_chord(start_chord)
            if current_chord not in self.graph.nodes:
                raise ValueError(f"Chord {start_chord} tidak ditemukan dalam graf")

            progression = [str(current_chord)]
            genre_weights = self.genre_weights.get(genre, self.genre_weights['pop'])

            for _ in range(length - 1):
                neighbors = list(self.graph.neighbors(current_chord))
                if not neighbors:
                    break

                # Hitung bobot untuk setiap neighbor berdasarkan genre
                weighted_neighbors = []
                for neighbor in neighbors:
                    base_weight = self.graph[current_chord][neighbor].get('weight', 0.5)

                    # Kolaborasi bobot genre
                    transition_key = (current_chord.quality, neighbor.quality)
                    genre_weight = genre_weights.get(transition_key, 0.5)

                    final_weight = base_weight * (genre_weight**3)
                    weighted_neighbors.append((neighbor, final_weight))

                # Pilih chord berikutnya berdasarkan weighted random selection
                if weighted_neighbors:
                    neighbors_list = [item[0] for item in weighted_neighbors]
                    weights = [item[1] for item in weighted_neighbors]
                    total_weight = sum(weights)
                    if total_weight > 0:
                        probs = [w/total_weight for w in weights]
                        choice_idx = np.random.choice(len(neighbors_list), p=probs)
                        current_chord = neighbors_list[choice_idx]
                    else:
                        current_chord = neighbors_list[np.random.randint(len(neighbors_list))]

                progression.append(str(current_chord))

            return progression
        except Exception as e:
            print(f"Error generating progression: {e}")
            return [start_chord]
```

Figure 3.2.5 Code of `generate_progression()` function

This function is used to generate automatic chord progressions from the initial chord provided by the user. For each step in the progression, the system will search for neighbors of the current chord, then select the next chord based on a combined weight calculation between basic harmonic side weight (e.g., P or R transition); genre-specific

transition weight (e.g., jazz prefers minor and dim); and an adjustment in the form of the square of the genre weight to make the genre more dominant in the selection. Chord selection is performed using a probabilistic approach through the weighted random selection method. This enables the system to generate chord progressions that remain consistent with the characteristics of the selected genre.

```
class TonnetzGraph:
    def classify_genre(self, progression):
        if len(progression) < 2:
            return {genre: 0.0 for genre in self.genre_weights.keys()}

        genre_scores = {}
        for genre in self.genre_weights.keys():
            genre_scores[genre] = 0.0

        for i in range(len(progression) - 1):
            try:
                current = self._parse_chord(progression[i])
                next_chord = self._parse_chord(progression[i + 1])
                transition = (current.quality, next_chord.quality)

                for genre, weights in self.genre_weights.items():
                    genre_scores[genre] += weights.get(transition, 0.1)

            except Exception:
                continue

        total_transitions = len(progression) - 1
        if total_transitions > 0:
            for genre in genre_scores:
                genre_scores[genre] /= total_transitions

        return genre_scores
```

Figure 3.2.6 Code of classify_genre() function

The classify_genre() function is used to predict the genre of music from a chord progression that has been formed. The classification process is carried out by comparing each transition between chords in the progression with the transition weights of each genre. The final result of this function is a dictionary containing match scores for each genre, which indicates how close the progression is to the harmonic characteristics of each genre.

```
class TonnetzGraph:
    def visualize_graph(self, highlight_progression=None):
        plt.figure(figsize=(15, 12))
        pos = nx.spring_layout(self.graph, k=3, iterations=50)
        nx.draw_networkx_edges(self.graph, pos, alpha=0.3, edge_color='gray')
        node_colors = []
        for node in self.graph.nodes():
            if node.quality == 'major':
                node_colors.append('lightblue')
            elif node.quality == 'minor':
                node_colors.append('lightcoral')
            else: # diminished
                node_colors.append('lightgreen')
        nx.draw_networkx_nodes(self.graph, pos, node_color=node_colors, node_size=800, alpha=0.8)
        labels = [node: str(node) for node in self.graph.nodes()]
        nx.draw_networkx_labels(self.graph, pos, labels, font_size=8)
        if highlight_progression:
            try:
                prog_nodes = [self._parse_chord(chord) for chord in highlight_progression]
                nx.draw_networkx_nodes(self.graph, pos, nodelist=prog_nodes,
                                     node_color='yellow', node_size=1000, alpha=0.7)
                for i in range(len(prog_nodes) - 1):
                    if self.graph.has_edge(prog_nodes[i], prog_nodes[i+1]):
                        nx.draw_networkx_edges(self.graph, pos,
                                              edges=[(prog_nodes[i], prog_nodes[i+1])],
                                              edge_color='red', width=4)
            except Exception:
                pass

        plt.title("Tonnetz Graph Representation")
        plt.axis('off')
        plt.tight_layout()
        plt.show()
```

Figure 3.2.7 Code of visualize_graph() function

The above function displays a visualization of the Tonnetz graph in planar form using the matplotlib library. The chords in the progression are visually marked with yellow (nodes) and red lines (transition sides), while the other nodes are distinguished based on chord type: light blue (major), pink (minor), and light green (diminished).

```
class TonnetzGraph:
    def analyze_progression(self, progression):
        print(f"Chord Progression: {progression}")
        print(f"Length: {len(progression)} chords")

        # Classify genre
        genre_scores = self.classify_genre(progression)
        print("\nGenre Classification:")

        # Sort genre scores
        sorted_genres = sorted(genre_scores.items(), key=lambda x: x[1], reverse=True)
        for genre, score in sorted_genres:
            print(f"{genre.capitalize()}: {score:.3f}")

        # Analyze transits
        print("\nChord Transitions:")
        for i in range(len(progression) - 1):
            try:
                current = self._parse_chord(progression[i])
                next_chord = self._parse_chord(progression[i + 1])

                if self.graph.has_edge(current, next_chord):
                    edge_data = self.graph[current][next_chord]
                    transform = edge_data.get('transform', 'Unknown')
                    weight = edge_data.get('weight', 0.0)
                    print(f"({progression[i]} -> {progression[i+1]}) (Transform: {transform}, Weight: {weight:.2f})")
            except Exception:
                print(f"({progression[i]} -> {progression[i+1]}) (Parse error)")
```

Figure 3.2.8 Code of analyze_progression() function

The analyze_progression() function is used to display a summary of the generated progression, including the type of transition between chords (e.g., P, R, L), its harmonic weight, and the genre classification results. This analysis helps users understand the harmonic structure of the progression that has been formed.

IV. TESTING

A. Testing Tonnetz as Planar Graph

```
# Uji keplanaran
is_planar, _ = nx.check_planarity(G)

# Tampilkan hasil
if is_planar:
    print("✓ Graf Tonnetz lokal (triad) ini adalah planar.")
    pos = nx.planar_layout(G)
    plt.figure(figsize=(8, 6))
    nx.draw(G, pos, with_labels=True, node_color='lightgreen', node_size=1000, font_size=12)
    plt.title("Subset Tonnetz sebagai Graf Planar")
    plt.show()
else:
    print("✗ Graf ini tidak planar.")
```

Figure 4.1.1 Checking if the tonnetz is planar graph

Based on the code created in the implementation section and using is_planar function, the code was executed and the following results were obtained that Tonnetz is a planar graph.

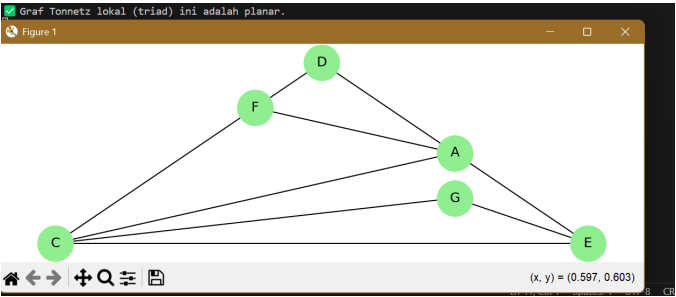


Figure 4.1.2 Results of checking using Python

After forming the Tonnetz graph, the next testing step can use Euler's formula by calculating the number of vertices (V), the number of edges (E), and the number of regions (F) bounded by the triad triangle. As a simple illustration: Suppose the Tonnetz graph has:

- V = 12 vertex (12 pitch classes)
- E = 30 edges (harmonic relations between pitches)

- $F = 20$ areas (triad triangles and outer regions)
- Therefore, tested using Euler's characteristic:

$$V - E + F = 12 - 30 + 20 = 2$$

The calculation result satisfies Euler's characteristic for a simple connected planar graph, as seen below:

$$V - E + F = 2$$

Thus, the Tonnetz graph structure constructed is a planar graph.

To further validate the test, Euler's inequality for simple planar graphs is also applied:

$$E \leq 3V - 6 \Rightarrow 30 \leq 3(12) - 6 = 30$$

Since the inequality is satisfied, this graph is a valid candidate as a planar graph.

Additionally, a test is performed using Kuratowski's theorem. According to Kuratowski's theorem, a graph is not planar if and only if it contains a subgraph isomorphic to:

- K_5 : a complete graph with 5 vertex
- $K_{3,3}$: a complete bipartite graph with two parts, each with 3 vertex

An examination of the local structure of the Tonnetz graph revealed no configurations of vertices and edges forming subgraphs isomorphic to those two graph. This can be visually inspected from the representation of adjacent triangles, with no vertices of degree 4 or higher being fully connected. Thus, based on Kuratowski's Theorem, the constructed Tonnetz graph is not a non-planar graph, reinforcing the conclusion that Tonnetz can be represented as a planar graph.

B. Generating Harmonic Chord Progressions

On this testing phase, the main() function is used to run the system interactively through the terminal interface. The program first creates a TonnetzGraph object that represents the planar graph of all chord combinations based on three main qualities: major, minor, and diminished. The initialization results show that the graph was successfully formed with a total of 36 nodes (12 notes multiplied by 3 chord types) and 60 edges connecting the chords based on harmonic transformations such as parallel (P), relative (R), leading-tone (L), fifth (V), and fourth (IV).

```
# testing
def main():
    print("=== TONNETZ GRAPH IMPLEMENTATION ===")
    print("Implementasi Tonnetz sebagai Graf Planar untuk Generasi Progresi Chord\n")

    tonnetz = TonnetzGraph()

    print(f"Graf berhasil dibuat dengan {tonnetz.graph.number_of_nodes()} nodes dan {tonnetz.graph.number_of_edges()} edges")

    print("\n=== TESTING ===")
    while True:
        try:
            start = input("\nMasukkan chord awal (atau 'quit' untuk keluar): ").strip()
            if start.lower() == 'quit':
                break

            length = int(input("Masukkan panjang progresi: "))
            genre = input("Masukkan genre (jazz/rock/pop/hiphop/classical): ").strip().lower()

            if genre not in tonnetz.genre_weights:
                genre = 'pop'
                print(f"Genre tidak dikenali, menggunakan 'pop' sebagai default")

            progression = tonnetz.generate_progression(start, length, genre)
            print(f"Generated Progression: {' -> '.join(progression)}")

            viz = input("\nVisualisasi graf? (y/n): ").strip().lower()
            if viz == 'y':
                tonnetz.visualize_graph(progression)

        except KeyboardInterrupt:
            break
        except Exception as e:
            print(f"Error: {e}")

    print("\nTerima kasih telah menggunakan Tonnetz Graph Implementation!")
```

Figure 4.2.1 Main program to display the generated results

Once the graph is formed, the user is prompted to enter input in the form of an initial chord, the desired progression length, and the music genre to be used as a weighting reference.

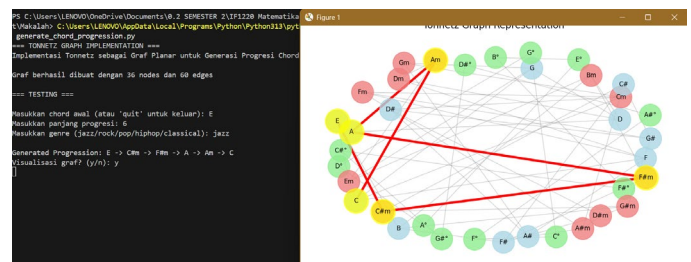


Figure 4.2.2 Testing result

For example, when the user enters the initial chord E, a progression length of 6, and selects the jazz genre, the system generates the following progression sequence: $E \rightarrow C\#m \rightarrow F\#m \rightarrow A \rightarrow Am \rightarrow C$. This result reflects the characteristics of jazz, which is known for its complex chord progressions and tonal shifts that are not always linear. The transition from major to minor chords, then to another minor chord, and shifting to the major subdominant and parallel minor demonstrates the effective application of the genre weighting designed for jazz. After the progression is displayed, the system also offers a graph visualization option. If selected, the system will display a complete graph containing all chords as nodes, with different colors for each type (major, minor, diminished). The resulting progression path is then highlighted in yellow for the nodes and with a thick red line for the transitions, allowing users to directly observe how the chord progression flows within the planar Tonnetz structure. This visualization reinforces understanding of harmonic structure and the effectiveness of traversal in generating progressions based on the selected genre.

V. CONCLUSION

This paper demonstrates that the Tonnetz structure can be effectively represented as a planar graph consisting of major, minor, and diminished chord nodes, as well as edges representing harmonic relationships between chords. By utilizing fundamental graph principles such as Neo-Riemannian transformations (P, R, L, V, and IV) and planar graph theory (Euler's formula and Kuratowski's theorem), the Tonnetz structure was successfully tested and confirmed as a valid simple planar graph.

The system implementation automatically generates chord progressions based on user input such as the initial chord, progression length, and chosen music genre. Chord selection is performed using weighted random traversal, where weights are determined based on harmonic proximity and genre preferences. Testing results show that the system can generate chord progressions that are not only musical and coherent but also reflect the characteristics of the chosen genre, such as complexity in jazz or balance in pop.

The graph visualization also successfully displays the progression path interactively, making it easier for users to understand the harmonic flow that occurs. Therefore, this approach demonstrates that graph theory from discrete mathematics can be practically utilized to support the automatic, systematic, and adaptive generation of digital music across different musical styles.

SOURCE CODE AT GITHUB

<https://github.com/alyanrrhma/Makalah-Matdis.git>

VIDEO LINK AT YOUTUBE

<https://bit.ly/ytbmakalahmatdis81>

ACKNOWLEDGMENT

The author gratefully acknowledges the blessings and guidance of Allah SWT, which provided the strength and guidance to complete this paper titled "*Tonnetz as a Planar Graph Representation for Generating Harmonic Chord Progressions in Songs*". Deep appreciation is also given to Bapak Dr. Rinaldi Munir, M.T., the lecturer of the Discrete Mathematics course for the Even Semester of 2024/2025, Class 01, whose insights and teachings a crucial foundation for this work. As the end, the author sincerely apologizes for any mistakes or inaccuracies that may exist in this paper.

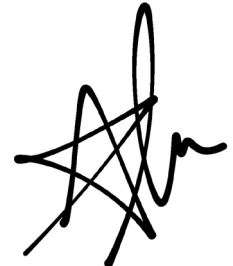
REFERENCES

- [1] Munir, Rinaldi. 2024. "Graf (Bag. 1)". <https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf>. Accessed 17 June 2025.
- [2] Munir, Rinaldi. 2024. "Graf (Bag. 2)". <https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf>. Accessed 17 June 2025.
- [3] Munir, Rinaldi. 2024. "Graf (Bag. 3)". <https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/22-Graf-Bagian3-2024.pdf>. Accessed 17 June 2025.
- [4] D. Tymoczko, *A Geometry of Music: Harmony and Counterpoint in the Extended Common Practice*. Oxford University Press, 2011. <https://global.oup.com/academic/product/a-geometry-of-music-9780195336672>. Accessed 19 June 2025.
- [5] M. Hewitt, "Musical Chords, Sets, and Graph Theory," *Journal of Mathematics and Music*, vol. 3, no. 3, pp. 117–135, 2009.
- [6] R. Parncutt, "A multi-level tonal interval space for modelling pitch relatedness and musical consonance," *Journal of New Music Research*, vol. 35, no. 2, pp. 145–172, 2006. https://www.researchgate.net/publication/303595116_A_multi-level_tonal_interval_space_for_modelling_pitch_relatedness_and_musical_consonance. Accessed 19 June 2025.
- [7] M. Numer, *A Link between Mathematics and Music: The Tonnetz*, Wooster College, 2025. <https://wooster.edu/wp-content/uploads/2025/04/Marissa-Numer-Poster.pdf>. Accessed 19 June 2025.

PERSONAL STATEMENT

I hereby declare that the paper I have written is my own work, not an adaptation or translation of someone else's paper, and not plagiarism.

Bandung, 20 Juni 2025



Alya Nur Rahmah 13524081