Graph Theory in Euler's Tonnetz: Applications in Composition and Harmonic Analysis

Shanice Feodora Tjahjono 13523097[1,2]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13523097@mahasiswa.itb.ac.id, [2]xianfeodora@gmail.com

***Abstract*— The Tonnetz, first described by Leonhard Euler, is a geometric representation of musical intervals used and incorporated by composers and musicians, especially in modern times. By bridging graph theory and music theory, this paper explores the Tonnetz as a graph structure, where vertices can represent pitch classes, whereas edges can represent certain intervals. Applications of graph-theoretical methods to harmonic analysis can be found in contemporary compositions thanks to the Tonnetz, showcasing how such approaches provide deeper insight into studying harmonic coherence and dissonance resolutions. We can also utilize the Tonnetz to study harmony in classical pieces, even if they were composed before the Tonnetz was founded. By integrating graph theory with Euler's Tonnetz, this study aims to contribute to the growing field of mathematical music theory and highlight the interdisciplinary potential of computational approaches in musicology.**

Keywords—Tonnetz, application of graph theory, composition, harmonic analysis

I. Introduction

Graph theory dates to when Leonard Euler solved the notable problem about the seven bridges of Königsberg in a 1736 paper [1]. In it, he proved that the problem had no solution. Many developments in graph theory have occurred since then - it is currently one of the most active fields of mathematics. Graph theory studies the relationships between objects, represented as nodes, or vertices, connected by edges. Initially studied in mathematics and computer science, it has become an important tool in various fields, such as chemistry, genetics, linguistics, geography, architecture, and music. Its ability to model complex relationships in a structured and visual manner makes it a powerful tool to explore intricate systems.

One such application is the Tonnetz, also developed by Euler. The Tonnetz maps pitch classes, triads, and their relationships [2]. It represents harmonic structures visually, making it useful for examining harmonic progressions and voice leading. Pitch classes or tones are represented as nodes, with their relationships or intervals represented as edges. These representations align gracefully with the theory of unweighted graphs, allowing a more formal study of harmony.

By understanding the Tonnetz and its connection to graph theory, such fusion between mathematics and music enables us to enhance our understanding of concepts in harmony, dissonance, and voice leading, providing valuable insights for composition, performance, or enjoyment of musical practices. By creating a model of the Tonnetz as a graph, concepts such as adjacency, cycles, and shortest paths can be demonstrated to enhance understanding of harmonic progressions and voice leading. To exemplify these principles, we developed a program that applies graph-theoretical methods to the Tonnetz, enabling computational analysis of harmonic structures.

This paper also presents its applications by analyzing contemporary compositions. By examining how composers and music aficionados have benefited from the Tonnetz or related concepts, we aim to demonstrate the practical and theoretical potential of this interdisciplinary approach.

II. Graph Theory

A. Definition

A graph is a structure consisting of sets of nodes (or vertices) and edges that connect two nodes. Graphs are used to model connections or relationships between objects [3].

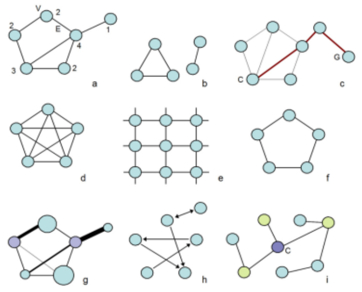


Figure 1. Visual representations of graphs

Source: https://www.researchgate.net/figure/sual-representations-of-graphs-Graph-theory-is-a-branch-of-discrete-mathematics-and-is_fig1_343610452

B. Nodes and Edges

Nodes, or vertices, represent the objects in a graph, whereas edges represent the connections or relationships between nodes [4]. If a node A is connected to node B with an edge, we can say that A and B are neighbors.

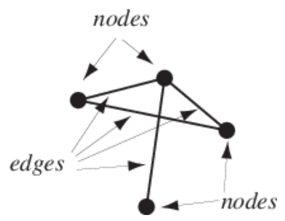


Figure 2. Nodes and Edges
Source: https://mathworld.wolfram.com/GraphEdge.html

C. Weighted and Unweighted Graphs

In graph theory, graphs can be categorized based on the existence of weights in their edges. Weights represent the characteristic or value between two nodes. For weighted graphs, every edge has a weight that represents a value, such as distance, price, or the intensity of the relationship between two nodes. As for unweighted graphs, every edge is considered to have the same weight, therefore all connections between nodes are equal [5].

D. Directed and Undirected Graphs

Graphs can also be categorized based on the presence of direction in edges. In directed graphs, each edge has a specific direction that indicates the relationship from one node to another. These edges are often represented by arrows [5]. An example of its application is a transportation network graph, where the direction indicates a one-way road between two locations.

In undirected graphs, edges have no direction, so the relationship between two nodes is symmetrical [5]. An example of its application is a graph in a social network, where we can consider the relationship between individuals as reciprocal.

E. Adjacency List

An adjacency list is one of several methods to model graphs. Each node is represented as an element in the data structure, and each element contains a list of other nodes connected to it [6]. Advantages of this method include efficiency in memory usage for sparse graphs and ease in iterating over the neighbors of a particular node.

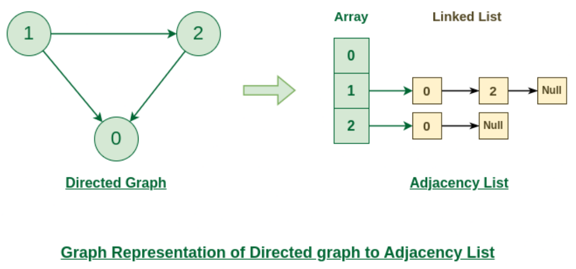


Figure 3. Comparison Between a Graph and Its Adjacency List
Source: https://www.geeksforgeeks.org/adjacency-list-meaning-definition-in-dsa/

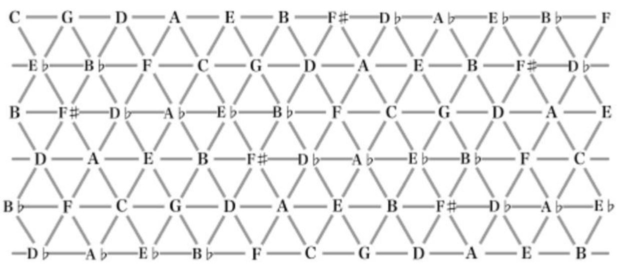
F. Breadth-First Search (BFS)

Breadth-First Search is a graph traversal algorithm that explores all nodes at a given level before moving to the next level. This algorithm is commonly used to find the shortest path in an unweighted graph, determine whether the graph is fully connected, and find the nearest node in a hierarchal structure. The BFS process takes place by beginning from the starting node, then uses a queue structure to keep track of nodes to be explored consequently [7].

III. The Tonnetz

The Tonnetz is a conceptual structure in music theory used to represent harmonic relationships geometrically [10]. This concept was first described in the 18th century by Leonhard Euler, a Swiss mathematician and physicist, in his work, "Speculum Musicum". Euler used the Tonnetz to illustrate tonal relationships in harmonic structure [8]. Later, Hugo Riemann developed this concept in the 19th century, integrating the Tonnetz with harmony transformation, which became the basis for the Neo-Riemannian Theory (NRT) [9]. Throughout the history of music since, Tonnetz has been used to visualize relationships between harmonic progression and tonal changes.

The Tonnetz is represented as a lattice structure that illustrates the relationship between tones. The geometric arrangement of the Tonnetz allows for visual analysis of pitch relationships, making it easier to identify harmonic patterns and tonal modulations [10].


C G D A E B F♯ D♭ A♭ E♭ B♭ F
E♭ B♭ F C G D A E B F♯ D♭
B F♯ D♭ A♭ E♭ B♭ F C G D A E
D A E B F♯ D♭ A♭ E♭ B♭ F C
B♭ F C G D A E B F♯ D♭ A♭ E♭
D♭ A♭ E♭ B♭ F C G D A E B
Figure 4. The Tonnetz
Source: J. C. Hart, "Isochords: Visualizing Structure in Music," Proceedings of the IEEE Visualization 2003

(VIS'03), Seattle, WA, USA, 2003, pp. 621-628.

IV. Discussion

A. Modelling Tonnetz as a Graph

The Tonnetz can be represented as an undirected graph where the tones in a chromatic scale are represented by nodes and the edges represent the intervals between tones. Through this approach, the Tonnetz can be modeled as an adjacency list with its nodes representing the 12 chromatic tones. Each node is connected to every other node by a certain harmonic interval, calculated using modular arithmetic to preserve the cyclical nature of the chromatic scale. The Tonnetz can also be interpreted as a graph with a dual graph, where nodes represent chords, and edges connect chords that share one or more tones. This dual graph provides an additional perspective in understanding the relationships between chords in the context of musical harmony.

B. Applications in Composition and Harmonic Analysis

Interactive composition tools such as PaperTonnetz enables composers to draw harmonic paths on Tonnetz representations on paper. This tool combines an interactive interface for real-time harmonic progression listening with Tonnetz graph representation. PaperTonnetz supports harmonic exploration in contemporary music by utilizing the graph's ability to illustrate the relationships between notes and chords [11].

Neo-Riemannian transformations, such as parallel (P), relative (R), and nearest-tone exchange (L), use Tonnetz structures to model the movement of triad chords efficiently. These transformations optimize voicing, the movement between tones with the minimum distance. This application has helped in the deeper analysis of harmonic relationships, both in classical and contemporary compositions [12]. Neo-Riemannian Theory (NRT) often uses Tonnetz to analyze compositions that focus on transformations between major and minor triads.

Examples of analysis in classical compositions include Schubert's and Dvořák's works. Pieces such as "Der Jüngling und der Tod" (D. 545) and "Trost" (D. 523) have been the subject of Neo-Riemannian analysis. The studies in "Two Neo-Riemannian Analyses" [13] emphasize the transformational relationships between triads and the nature of voice leading in these works. Another example can be found in the study "An Analysis of Dvořák's Symphony No. 6, II, Using Neo-Riemannian Transformation Techniques" [14], where the second movement of Dvořák's symphony is analyzed using NRT to understand the complex harmonic transformations and voice leading in the symphony's structure.

As mentioned previously, NRT has also been used to analyze contemporary compositions. Wayne Shorter's jazz compositions using harmonic transformations have been analyzed with NRT, providing insight into the unique harmonic language he uses [15].

V. Experiment

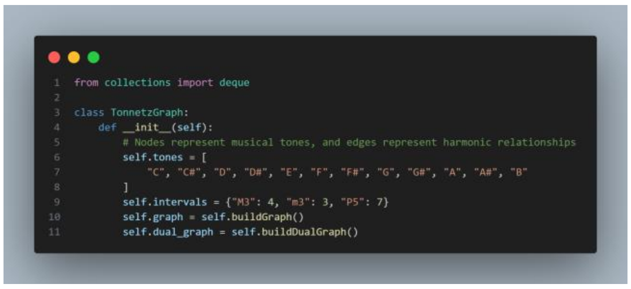
This experiment was designed to see how graph theory can be implemented in the Tonnetz, especially its potential applications in composition and harmonic analysis. Although this program doesn't cover the concept of the Tonnetz to its full extent, its aim remains to show the connection between graph theory and the Tonnetz. By modeling such a structure through programming, this experiment aims to,

1. Visualize harmonic relationships effectively,
2. Implement algorithms to analyze tonal relationships and
3. Provide insight into graph-based learning in music theory.

In this experiment, *Python* is utilized to build the graph structure and execute its main operations, such as finding and recommending chord progressions and identifying intervals.

A. Program Design

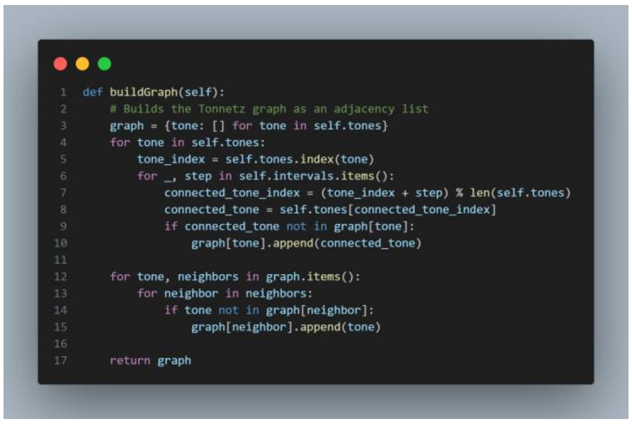
As a *Python* implementation of the Tonnetz, this program utilizes graph structure, where nodes represent tones and edges represent the intervals between tones. This program also adopts the dual graph feature of the Tonnetz, where nodes represent chords and edges represent shared tones between chords. Implementing Tonnetz's dual graph enables analysis, such as finding the shortest possible chord progression between two tones using the BFS algorithm. Another feature added to this program is generating a 2-5-1 progression, a technique commonly found in jazz. Due to the addition of this feature, we can see how the Tonnetz and graph theory can be found relevant in notable music techniques that are still frequently applied to this day. The program also has a feature to identify intervals between two tones, such as the major third, the minor third, or the perfect fifth. This program offers a simple yet interactive interface where users can explore harmonic structures computationally.



```python
from collections import deque

class TonnetzGraph:
    def __init__(self):
        # Nodes represent musical tones, and edges represent harmonic relationships
        self.tones = [
            "C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A", "A#", "B"
        ]
        self.intervals = {"M3": 4, "m3": 3, "P5": 7}
        self.graph = self.buildGraph()
        self.dual_graph = self.buildDualGraph()
```

Figure 5. Constructing the Tonnetz Graph

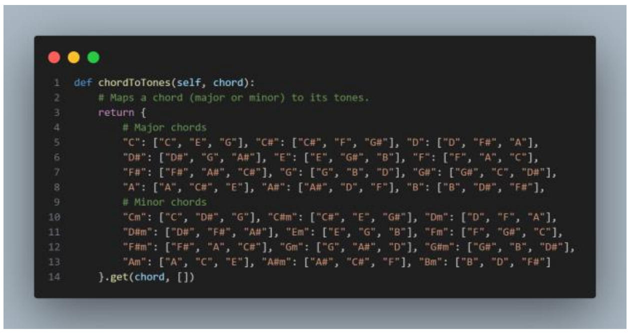
The class TonnetzGraph is the main representation of the Tonnetz graph. Its function is to build a graph that represents the intervals between tones in the chromatic scale, in accordance to the concept of the Tonnetz. The figure also shows the constructor function of the program, where it initializes all attributes and data structures needed to represent the Tonnetz graph.



```python
def buildGraph(self):
    # Builds the Tonnetz graph as an adjacency list
    graph = {tone: [] for tone in self.tones}
    for tone in self.tones:
        tone_index = self.tones.index(tone)
        for _, step in self.intervals.items():
            connected_tone_index = (tone_index + step) % len(self.tones)
            connected_tone = self.tones[connected_tone_index]
            if connected_tone not in graph[tone]:
                graph[tone].append(connected_tone)

    for tone, neighbors in graph.items():
        for neighbor in neighbors:
            if tone not in graph[neighbor]:
                graph[neighbor].append(tone)

    return graph
```

Figure 6. Structuring the Tonnetz Graph as an Adjacency List and Ensuring it is Undirected

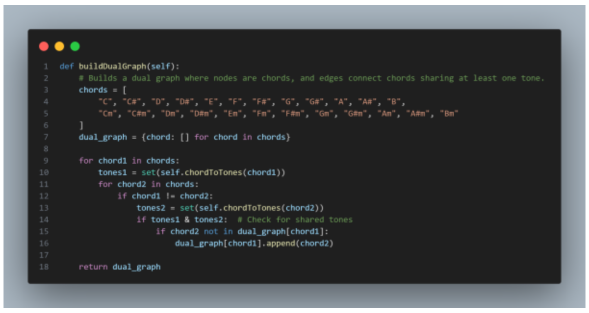
This function ensures that the constructed Tonnetz graph is represented as an undirected graph. To do so, it iterates through each node and its neighbors in the adjacency list. For each connection from node A to B, the function checks whether a connection from B to A already exists. If not, this connection is added to the adjacency list.



```python
def chordToTones(self, chord):
    # Maps a chord (major or minor) to its tones.
    return {
        # Major chords
        "C": ["C", "E", "G"], "C#": ["C#", "F", "G#"], "D": ["D", "F#", "A"],
        "D#": ["D#", "G", "A#"], "E": ["E", "G#", "B"], "F": ["F", "A", "C"],
        "F#": ["F#", "A#", "C#"], "G": ["G", "B", "D"], "G#": ["G#", "C", "D#"],
        "A": ["A", "C#", "E"], "A#": ["A#", "D", "F"], "B": ["B", "D#", "F#"],
        # Minor chords
        "Cm": ["C", "D#", "G"], "C#m": ["C#", "E", "G#"], "Dm": ["D", "F", "A"],
        "D#m": ["D#", "F#", "A#"], "Em": ["E", "G", "B"], "Fm": ["F", "G#", "C"],
        "F#m": ["F#", "A", "C#"], "Gm": ["G", "A#", "D"], "G#m": ["G#", "B", "D#"],
        "Am": ["A", "C", "E"], "A#m": ["A#", "C#", "F"], "Bm": ["B", "D", "F#"]
    }.get(chord, [])
```

Figure 7. Listing All Chords with their Respective Tones

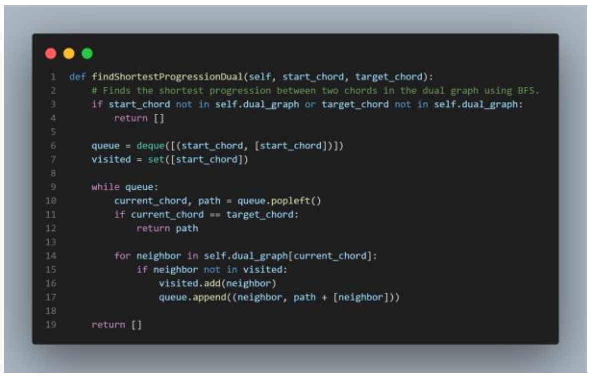
This function uses a dictionary to map a chord (major or minor) to its constituent tones. If the chord given as input is not found in the dictionary, the function returns an empty list.



```python
def buildDualGraph(self):
    # Builds a dual graph where nodes are chords, and edges connect chords sharing at least one tone.
    chords = [
        "C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A", "A#", "B",
        "Cm", "C#m", "Dm", "D#m", "Em", "Fm", "F#m", "Gm", "G#m", "Am", "A#m", "Bm"
    ]
    dual_graph = {chord: [] for chord in chords}

    for chord1 in chords:
        tones1 = set(self.chordToTones(chord1))
        for chord2 in chords:
            if chord1 != chord2:
                tones2 = set(self.chordToTones(chord2))
                if tones1 & tones2:  # Check for shared tones
                    if chord2 not in dual_graph[chord1]:
                        dual_graph[chord1].append(chord2)

    return dual_graph
```

Figure 8. Constructing the Tonnetz Dual Graph

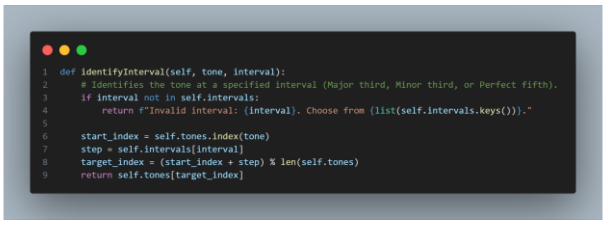
This function constructs a dual graph where nodes represent chords, and edges connect chords that share at least one tone. It first initializes an empty graph for all major and minor chords. For each pair of chords, the function checks whether they share one or more common tones using set operations. If a shared note is found and the pair is not yet connected, chord2 is added as a neighbor of chord1. The function returns an adjacency list, modeling the tone-based relationships between chords in a Tonnetz graph.



```python
def findShortestProgressionDual(self, start_chord, target_chord):
    # Finds the shortest progression between two chords in the dual graph using BFS.
    if start_chord not in self.dual_graph or target_chord not in self.dual_graph:
        return []

    queue = deque([(start_chord, [start_chord])])
    visited = set([start_chord])

    while queue:
        current_chord, path = queue.popleft()
        if current_chord == target_chord:
            return path

        for neighbor in self.dual_graph[current_chord]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, path + [neighbor]))

    return []
```

Figure 9. Shortest Progression Feature

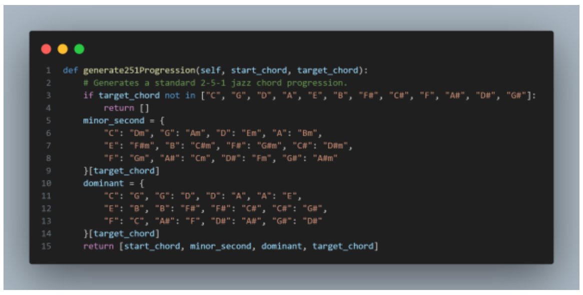
This function utilizes the BFS algorithm to find the shortest progression path between two chords by using the dual graph as shown in Figure 8. It starts by verifying whether the start chord and target chord exist in the dual graph. If not, the function returns an empty list. Then, a queue is initialized to store the chord and path pairs, while a set is used to keep track of the visited nodes. BFS is executed by traversing the neighbors of each chord until the target chord is found, and the shortest path is returned.



```python
def identifyInterval(self, tone, interval):
    # Identifies the tone at a specified interval (Major third, Minor third, or Perfect fifth).
    if interval not in self.intervals:
        return f"Invalid interval: {interval}. Choose from {list(self.intervals.keys())}."

    start_index = self.tones.index(tone)
    step = self.intervals[interval]
    target_index = (start_index + step) % len(self.tones)
    return self.tones[target_index]
```

Figure 10. Interval Identification Feature

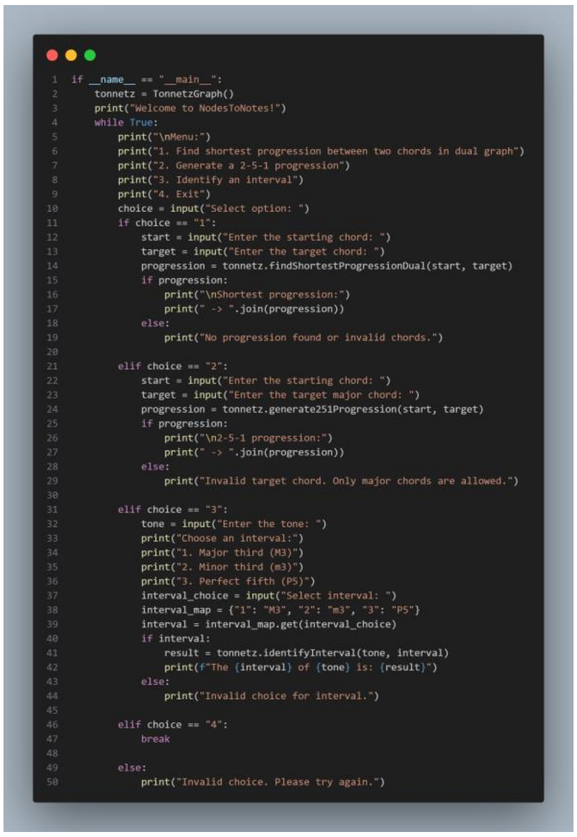
This function identifies the tone at a specified interval from a starting note, both inputted by the user. It first checks whether the requested interval is valid by matching it against a list of available intervals. If it is valid, the function calculates the index of the starting note in the list of notes, adds a step value according to the interval, and uses modular arithmetic to ensure the index remains within the chromatic scale range. The note at the target index is then returned as the result.



```python
def generate251Progression(self, start_chord, target_chord):
    # Generates a standard 2-5-1 jazz chord progression.
    if target_chord not in ["C", "G", "D", "A", "E", "B", "F#", "C#", "F", "A#", "D#", "G#"]:
        return []
    minor_second = {
        "C": "Dm", "G": "Am", "D": "Em", "A": "Bm",
        "E": "F#m", "B": "C#m", "F#": "G#m", "C#": "D#m",
        "F": "Gm", "A#": "Cm", "D#": "Fm", "G#": "A#m"
    }[target_chord]
    dominant = {
        "C": "G", "G": "D", "D": "A", "A": "E",
        "E": "B", "B": "F#", "F#": "C#", "C#": "G#",
        "F": "C", "A#": "F", "D#": "A#", "G#": "D#"
    }[target_chord]
    return [start_chord, minor_second, dominant, target_chord]
```

Figure 11. 2-5-1 Progression Feature

This function is used to generate a standard 2-5-1 jazz chord progression. The function begins by verifying whether the target chord is listed among the chords that support the 2-5-1 progression. The function returns an empty list otherwise. The function uses a predetermined table to identify the dominant and minor second chords for valid chords. These chords are then combined with the starting chord and the target chord to create a full 2-5-1 progression, with the starting chord displayed first, then the target chord's minor second, dominant, and tonic.



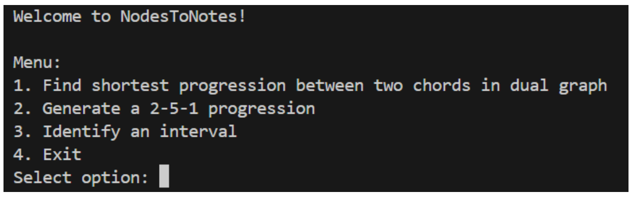
```python
if __name__ == "__main__":
    tonnetz = TonnetzGraph()
    print("Welcome to NodesToNotes!")
    while True:
        print("\nMenu:")
        print("1. Find shortest progression between two chords in dual graph")
        print("2. Generate a 2-5-1 progression")
        print("3. Identify an interval")
        print("4. Exit")
        choice = input("Select option: ")
        if choice == "1":
            start = input("Enter the starting chord: ")
            target = input("Enter the target chord: ")
            progression = tonnetz.findShortestProgressionDual(start, target)
            if progression:
                print("\nShortest progression:")
                print(" -> ".join(progression))
            else:
                print("No progression found or invalid chords.")

        elif choice == "2":
            start = input("Enter the starting chord: ")
            target = input("Enter the target major chord: ")
            progression = tonnetz.generate251Progression(start, target)
            if progression:
                print("\n2-5-1 progression:")
                print(" -> ".join(progression))
            else:
                print("Invalid target chord. Only major chords are allowed.")

        elif choice == "3":
            tone = input("Enter the tone: ")
            print("Choose an interval:")
            print("1. Major third (M3)")
            print("2. Minor third (m3)")
            print("3. Perfect fifth (P5)")
            interval_choice = input("Select interval: ")
            interval_map = {"1": "M3", "2": "m3", "3": "P5"}
            interval = interval_map.get(interval_choice)
            if interval:
                result = tonnetz.identifyInterval(tone, interval)
                print(f"The {interval} of {tone} is: {result}")
            else:
                print("Invalid choice for interval.")

        elif choice == "4":
            break

        else:
            print("Invalid choice. Please try again.")
```

Figure 12. Main Program

Users can interact with the Tonnetz graph's features using the main interface of the terminal-based program. User input, such as a starting and target chord, a starting note, or a certain interval, can be entered into the program and processed by the TonnetzGraph class's functions. The user has the option of finding the shortest progression between two chords, generating a 2-5-1 chord progression, or identifying the notes in a particular interval. Additionally, input validation is built into the program to ensure that only valid input is handled, displaying error messages if an invalid input is received. The purpose of this interface is to make exploring the Tonnetz's features more organized and interactive.

B. Testing

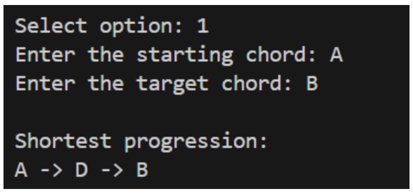
The testing phase evaluates the program's implementation of Tonnetz as a graph, focusing on the accuracy and functionality of its features. Testing is performed through various scenarios to ensure that features such as shortest progression, 2-5-1 progression generation, and interval identification correctly apply graph theory principles to model harmonic structures. This process ensures the program's reliability in using Tonnetz for computational harmonic analysis.



```
Welcome to NodesToNotes!

Menu:
1. Find shortest progression between two chords in dual graph
2. Generate a 2-5-1 progression
3. Identify an interval
4. Exit
Select option: 
```

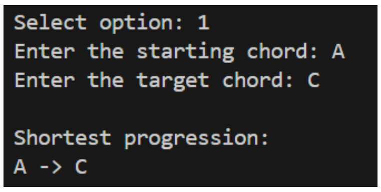
Figure 13. Menu

The figure above shows the display of the program when first initiaized. The user is given options to access features in NodesToNotes, a program made based on the Tonnetz graph.



```
Select option: 1
Enter the starting chord: A
Enter the target chord: B

Shortest progression:
A -> D -> B
```

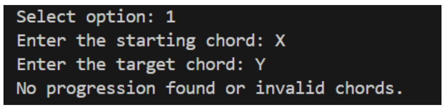
Figure 14. Valid Input for Shortest Progression Feature



```
Select option: 1
Enter the starting chord: A
Enter the target chord: C

Shortest progression:
A -> C
```

Figure 15. Valid Input for Shortest Progression Feature

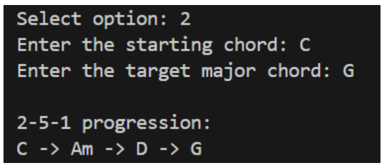
Figures 14 and 15 show the output of the shortest progression feature when the user enters valid input. Figure 14 displays a progression of three chords: the starting chord, a passing chord, and the target chord, whereas Figure 15 only displays the starting and target chord. The starting and target chords do not have any shared tones; hence, a passing chord that shares tones with both the starting and target chords is required. Conversely, the chords being entered in Figure 15 share a tone, making the progression from the starting chord to the target chord the shortest.



```
Select option: 1
Enter the starting chord: X
Enter the target chord: Y
No progression found or invalid chords.
```

Figure 16. Invalid Input for Shortest Progression Feature

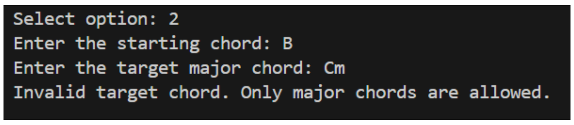
The program returns an error message if the chords entered aren't valid musical major or minor chords.



```
Select option: 2
Enter the starting chord: C
Enter the target major chord: G

2-5-1 progression:
C -> Am -> D -> G
```

Figure 17. Valid Input for 2-5-1 Progression Feature

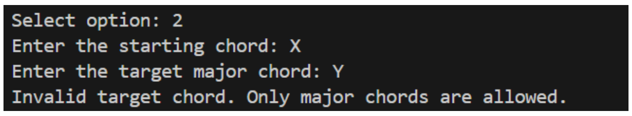
When the user enters valid input for this feature, the program returns a generated 2-5-1 progression, beginning with the starting chord, followed by the minor second, dominant, and tonic of the target major chord.



```
Select option: 2
Enter the starting chord: B
Enter the target major chord: Cm
Invalid target chord. Only major chords are allowed.
```

Figure 18. Minor Target Chord Input for 2-5-1 Progression Feature

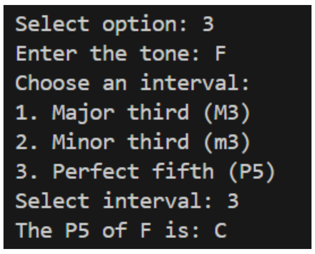
This program only accepts major chords as the target chord as minor target chords require diminished chords in standard 2-5-1 progression, which aren't accommodated by this program. Hence, the program returns a message that users can only enter major chords as the target chord.



```
Select option: 2
Enter the starting chord: X
Enter the target major chord: Y
Invalid target chord. Only major chords are allowed.
```

Figure 19. Invalid Input for 2-5-1 Progression Feature

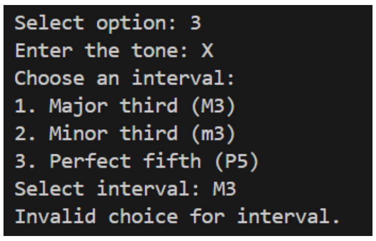
Similar to Figure 16, the program returns an error message if the chords entered aren't valid musical major or minor chords.



```
Select option: 3
Enter the tone: F
Choose an interval:
1. Major third (M3)
2. Minor third (m3)
3. Perfect fifth (P5)
Select interval: 3
The P5 of F is: C
```

Figure 20. Valid Input for Interval Identification Feature

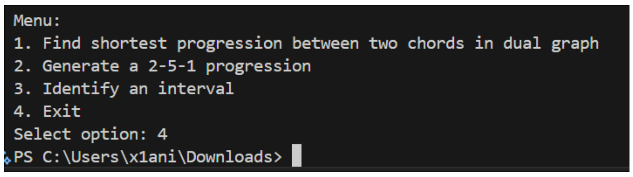
When the user enters valid input for this feature, the program identifies the tone at the specified interval relative to the input tone. In this example, the program calculates the perfect fifth (P5) of the tone F and returns the correct result, C.



```
Select option: 3
Enter the tone: X
Choose an interval:
1. Major third (M3)
2. Minor third (m3)
3. Perfect fifth (P5)
Select interval: M3
Invalid choice for interval.
```

Figure 21. Invalid Input for Interval Identification Feature

The program returns an error message if the user enters an invalid tone or interval.



```
Menu:
1. Find shortest progression between two chords in dual graph
2. Generate a 2-5-1 progression
3. Identify an interval
4. Exit
Select option: 4
PS C:\Users\x1ani\Downloads>
```

Figure 22. Exit

The program exits the program when the exit option is chosen.

C. Additional notes

Despite the program's ability to show fundamental connections between graph theory and the Tonnetz, there are still several aspects that can be improved upon to create richer understanding toward music theory and application possibilities.

The program still has several limitations. One of its examples includes how the model assumes that every harmonic interval is the same due to the unweighted characteristic of the graph. Real-world harmonic analysis can utilize weighted edges to represent varying degrees of harmonic tension.

Another limitation is that the model is restricted to major and minor chords. Other forms of chords, such as the diminished and augmented chords, or extended chords, such as sevenths and ninths, can also be identified through the Tonnetz but aren't modeled by the program.

Considering these limitations, improvements can be implemented in the future to make this prototype usable on a more versatile scale. The graph model can be modified to be weighted. That way, harmonic analysis can be executed more comprehensively. Secondly, more chord options can be added to the program to resemble the full capabilities of the Tonnetz. Third, the model could integrate GUI-based visualization of the Tonnetz, enabling users to see how the Tonnetz depicts tonal relationships and chord structures. In addition to that, the model could also provide interactive animations to depict harmonic transition and flow. Technical-wise, future developers could improve the efficiency of the pathfinding algorithm to support weighted paths and deeper harmonic exploration.

VI. CONCLUSION

This paper documents how Tonnetz concepts, essential to mathematical music theory, can be modelled and explored effectively using graph theory. The program combines music theory and computational analysis by using graph structures to visualize harmonic relationships and chord progressions, offering a potent groundwork for comprehending and visualizing harmonic structures.

In addition to its theoretical contributions, the presented implementation opens opportunities for cross-disciplinary applications, particularly in algorithmic or

AI-based tools in composition and music analysis. The ability to computationally model and manipulate harmonic structures could propel innovation in music education and analytical techniques, highlighting the synergy between mathematics, computer science, and the arts. These collaborations between fields birth the potential for further exploration and innovation at the boundaries of these disciplines.

VII. Acknowledgment

The author expresses their deepest gratitude to all lecturers of IF1220 Discrete Mathematics, especially Dr. Ir. Rinaldi Munir, M.T. as the lecturer of class K-01, for his constant guidance and expertise throughout the semester. The author also extends appreciation to the Bandung Institute of Technology for its resources and facilities, and to friends and family for their unwavering support during the writing of this paper.

References

[1] L. Euler, "Solutio problematis ad geometriam situs pertinentis" *Commentarii Academiae Scientiarum Petropolitanae*, vol. 8, pp. 128–140, 1736.
[2] L. Euler, "Tentamen novae theoriae musicae" Saint Petersburg: Imperial Academy of Sciences, 1739.
[3] R. Diestel, "Graph Theory" Springer, 2017.
[4] J. A. Bondy and U. S. R. Murty, "Graph Theory" Springer, 2008.
[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA: MIT Press, 2009.
[6] M. T. Goodrich and R. Tamassia, *Data Structures and Algorithms in Java*. Wiley, 2014.
[7] R. Sedgewick and K. Wayne, *Algorithms*. Addison-Wesley, 2011.
[8] L. Euler, *Speculum Musicum*. Basel, Switzerland: Birkhäuser, 1739.
[9] H. Riemann, *Handbuch der Harmonielehre*. Berlin, Germany: Breitkopf und Härtel, 1893.
[10] Tymoczko, D., "The Geometry of Musical Chords," *Science*, vol. 313, no. 5783, pp. 72–74, 2006. DOI: 10.1126/science.1126287.
[11] J. Garcia, L. Bigo, A. Spicher, and W. E. Mackay, "PaperTonnetz: Supporting Music Composition with Interactive Paper," *Extended Abstracts on Human Factors in Computing Systems*, Paris, France, Apr. 2013.
[12] F. Absil, "Neo-Riemannian transformations and the Tonnetz," Frans Absil's Website. [Online]. Available: https://www.fransabsil.nl/htm/tonnetz_riemannian_transformations.htm. Accessed: Jan. 5, 2025, 17:54.
[13] T. Brower, "Two Neo-Riemannian Analyses," *College Music Symposium*, vol. 45, 2005.
[14] M. Kennedy, "Analyzing Dvořák's Symphony No. 6: A Neo-Riemannian Perspective," *Honors Theses*, University of Southern Mississippi, 2016.
[15] S. B. P. Briginshaw, "A Neo-Riemannian Approach to Jazz Analysis," *Nota Bene: Canadian Undergraduate Journal of Musicology*, vol. 5, no. 1, Art. 5, 2012.

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, 7 Januari 2025

[illegible]

Shanice Feodora Tjahjono 13523097