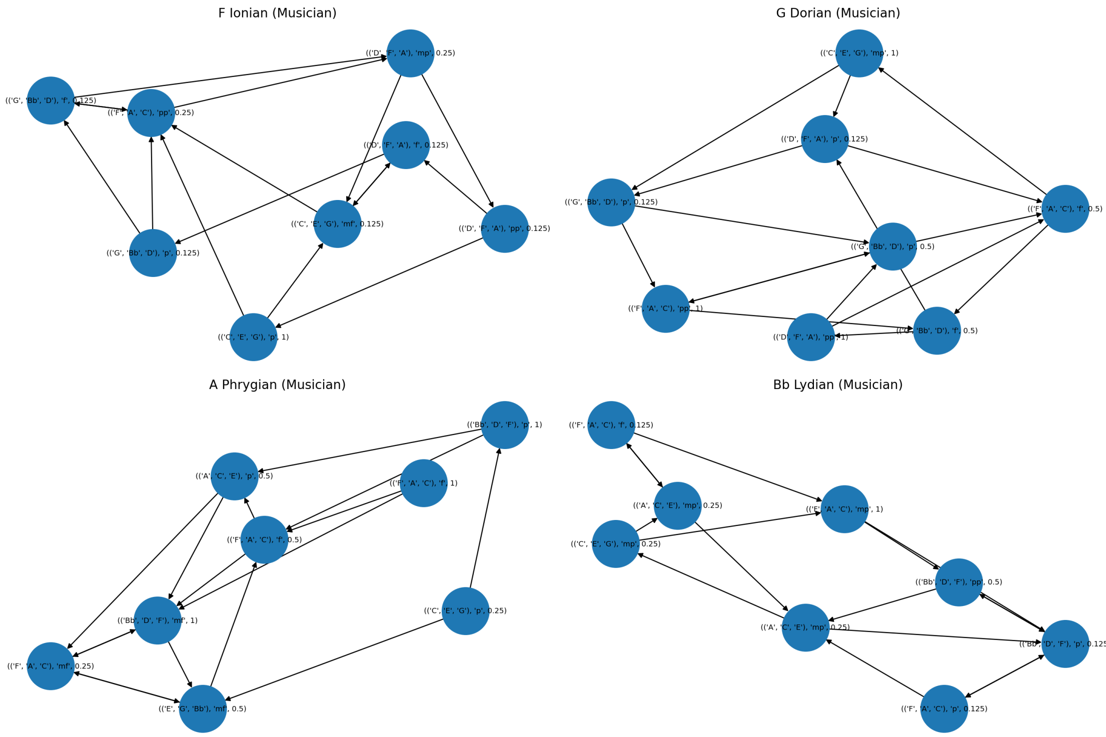
Quantum Music Composition

June 10, 2025

[25]:

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

# Define scales (modes)
modes = {
    "F Ionian": ["F", "G", "A", "Bb", "C", "D", "E"],
    "G Dorian": ["G", "A", "Bb", "C", "D", "E", "F"],
    "A Phrygian": ["A", "Bb", "C", "D", "E", "F", "G"],
    "Bb Lydian": ["Bb", "C", "D", "E", "F", "G", "A"],
}

# Chord qualities to build triads
chord_qualities = [
    [0, 2, 4],  # root, third, fifth in the scale
    [1, 3, 5],
    [2, 4, 6],
    [3, 5, 0],
    [4, 6, 1],
    [5, 0, 2],
    [6, 1, 3],
]

intensities = ["pp", "p", "mp", "mf", "f"]
durations = [1/8, 1/4, 1/2, 1]

# Generate 4 random graphs
random.seed(42)

graphs = []
for i, (mode_name, scale_notes) in enumerate(modes.items()):
    # 8 unique chords for the nodes
    nodes = []
    for n in range(8):
        chord_idx = random.choice(chord_qualities)
        chord = tuple([scale_notes[i % 7] for i in chord_idx])
        intensity = random.choice(intensities)
```

```
        duration = random.choice(durations)
        nodes.append((chord, intensity, duration))

    # Build directed graph
    G = nx.DiGraph()
    for idx, node in enumerate(nodes):
        G.add_node(idx, label=str(node))

    # Add at least 2 random outgoing edges per node
    for idx in range(8):
        targets = set()
        while len(targets) < 2:
            tgt = random.randint(0, 7)
            if tgt != idx:
                targets.add(tgt)
        for tgt in targets:
            G.add_edge(idx, tgt)
    graphs.append((mode_name, G, nodes))

# Plotting
fig, axes = plt.subplots(2, 2, figsize=(15, 10))
axes = axes.flatten()

for ax, (mode_name, G, nodes) in zip(axes, graphs):
    labels = nx.get_node_attributes(G, 'label')
    pos = nx.spring_layout(G, seed=10)
    nx.draw(G, pos, ax=ax, with_labels=True, labels=labels, node_size=2000,␣
 ↪font_size=7, arrows=True)
    ax.set_title(f"{mode_name} (Musician)")

plt.tight_layout()
plt.savefig("4-Musicians.pdf")
plt.show()
```



[ ]:

0.0.1 Quantum Music Composition via Szegedy Quantum Walk

[19]:

```python
import numpy as np
import networkx as nx
import music21 as m21
from tabulate import tabulate

# -- Parameters --
notes_by_musician = {
    1: ["F", "G", "A", "Bb", "C", "D", "E"],      # F Ionian
    2: ["G", "A", "Bb", "C", "D", "E", "F"],      # G Dorian
    3: ["A", "Bb", "C", "D", "E", "F", "G"],      # A Phrygian
    4: ["C", "D", "E", "F", "G", "A", "Bb"],      # C Mixolydian
}

intensities = ["pp", "p", "mp", "mf", "f"]
durations = [0.125, 0.25, 0.5, 1.0]  # quarter note = 1.0

num_nodes = 8  # per musician

# Simple function to create random chords from allowed notes
```

```python
def make_chord(notes):
    # Make triad chord: root + 2 others at +2 and +4 scale degrees mod 7
    root_idx = np.random.randint(0, len(notes))
    chord = [
        notes[root_idx],
        notes[(root_idx + 2) % len(notes)],
        notes[(root_idx + 4) % len(notes)],
    ]
    return tuple(chord)

# Build directed graph with random transitions for one musician
def build_musician_graph(musician_id):
    scale_notes = notes_by_musician[musician_id]
    nodes = []
    for _ in range(num_nodes):
        chord = make_chord(scale_notes)
        intensity = np.random.choice(intensities)
        duration = np.random.choice(durations)
        nodes.append((chord, intensity, duration))

    G = nx.DiGraph()
    for i, node in enumerate(nodes):
        G.add_node(i, data=node)

    # Each node has >= 2 outgoing edges to random distinct nodes (no self-loops)
    for i in range(num_nodes):
        targets = set()
        while len(targets) < 2:
            tgt = np.random.randint(0, num_nodes)
            if tgt != i:
                targets.add(tgt)
        for tgt in targets:
            G.add_edge(i, tgt)

    # Create row-stochastic transition matrix T
    T = np.zeros((num_nodes, num_nodes))
    for i in range(num_nodes):
        out_edges = list(G.successors(i))
        prob = 1.0 / len(out_edges)
        for j in out_edges:
            T[i, j] = prob

    return G, T

# Szegedy Walk Construction for one musician
def szegedy_walk(T, steps, initial_vertex=0):
    n = T.shape[0]
```

```python
# Build states |phi_u> = sum_v sqrt(T[u,v])|v>
phi = np.array([np.sqrt(T[u, :]) for u in range(n)])

# Build projectors A and B
A = np.zeros((n*n, n*n))
for u in range(n):
    ket_u = np.zeros(n)
    ket_u[u] = 1
    ket_phi_u = phi[u]
    proj = np.kron(np.outer(ket_u, ket_u), np.outer(ket_phi_u, ket_phi_u))
    A += proj
B = np.zeros((n*n, n*n))
for v in range(n):
    ket_v = np.zeros(n)
    ket_v[v] = 1
    ket_phi_v = phi[:, v]
    proj = np.kron(np.outer(ket_phi_v, ket_phi_v), np.outer(ket_v, ket_v))
    B += proj

# Reflection operators
I = np.eye(n*n)
R_A = 2 * A - I
R_B = 2 * B - I

# Walk operator
W = R_B @ R_A

# Initial state |psi_0> = |v_0> \otimes |phi_{v_0}>
ket_v0 = np.zeros(n)
ket_v0[initial_vertex] = 1
ket_phi_v0 = phi[initial_vertex]
psi = np.kron(ket_v0, ket_phi_v0)

# Iterate walk
for _ in range(steps):
    psi = W @ psi

# Measure first register: probability of each vertex u
probs = np.zeros(n)
for u in range(n):
    for v in range(n):
        idx = u * n + v
        probs[u] += np.abs(psi[idx]) ** 2
probs /= np.sum(probs)

# Sample vertex from distribution
sampled_vertex = np.random.choice(range(n), p=probs)
```

```
    return sampled_vertex

# Create a music21 stream for one musician given a list of vertex indices and␣
↪graph
def create_music_stream(graph, vertices, instrument_name="Piano"):
    stream = m21.stream.Stream()
    instr = getattr(m21.instrument, instrument_name)()
    stream.append(instr)

    for v in vertices:
        chord_notes, intensity, duration = graph.nodes[v]['data']
        chord_obj = m21.chord.Chord(chord_notes)
        chord_obj.duration = m21.duration.Duration(duration)
        # Map intensity string to midi velocity
        velocity_map = {"pp": 40, "p": 55, "mp": 70, "mf": 85, "f": 100}
        velocity = velocity_map.get(intensity, 70)
        chord_obj.volume.velocity = velocity
        stream.append(chord_obj)

    return stream

def print_sequence_table(graph, sequence, musician_id):
    table_data = []
    for i, v in enumerate(sequence):
        chord_notes, intensity, duration = graph.nodes[v]['data']
        chord_str = " ".join(chord_notes)
        table_data.append([i + 1, chord_str, intensity, duration])
    headers = ["Step", "Chord", "Intensity", "Duration"]
    print(f"\nMusician {musician_id} sequence:")
    print(tabulate(table_data, headers=headers, tablefmt="fancy_grid"))


# Modified generate_quantum_music to return sequences + graphs to print tables
def generate_quantum_music_with_sequences(steps_per_musician=1000):
    graphs = []
    sequences = []
    streams = []

    for musician in range(1, 5):
        G, T = build_musician_graph(musician)
        sequence = []
        current_vertex = 0
        for _ in range(steps_per_musician):
            sampled_vertex = szegedy_walk(T, steps=5,␣
↪initial_vertex=current_vertex)
            sequence.append(sampled_vertex)
```

```
            current_vertex = sampled_vertex

        graphs.append(G)
        sequences.append(sequence)

        stream = create_music_stream(G, sequence, instrument_name=["Piano",␣
↪"Violin", "Flute", "Clarinet"][musician-1])
        streams.append(stream)

    # Combine streams into a score
    score = m21.stream.Score()
    for s in streams:
        score.append(s)
    return score, graphs, sequences



score, graphs, sequences =␣
↪generate_quantum_music_with_sequences(steps_per_musician=1000)

score.write('midi', fp='My_Szegedy_music.mid')

# Now save each musician's sequence as PDF
#for i, (graph, seq) in enumerate(zip(graphs, sequences), start=1):
    #save_sequence_to_pdf(graph, seq, musician_id=i)
```

[19]: 'My_Szegedy_music.mid'

[ ]:

[ ]:

[ ]:

0.0.2 Quantum Music Composition via Coined Quantum Walk

[21]:
```
import numpy as np
import networkx as nx
import music21 as m21

# Allowed notes by musician (from your F Major modes)
notes_by_musician = {
    1: ["F", "G", "A", "Bb", "C", "D", "E"],      # F Ionian
    2: ["G", "A", "Bb", "C", "D", "E", "F"],      # G Dorian
    3: ["A", "Bb", "C", "D", "E", "F", "G"],      # A Phrygian
    4: ["C", "D", "E", "F", "G", "A", "Bb"],      # C Mixolydian
}
```

```
intensities = ["pp", "p", "mp", "mf", "f"]
durations = [0.125, 0.25, 0.5, 1.0]  # 1/8, 1/4, 1/2, whole note

num_nodes = 8  # per musician

# Generate random triad chord from notes for each musician
def make_chord(notes):
    root_idx = np.random.randint(len(notes))
    # triad with intervals +2 and +4 mod scale length
    chord = [
        notes[root_idx],
        notes[(root_idx + 2) % len(notes)],
        notes[(root_idx + 4) % len(notes)],
    ]
    return tuple(chord)

# Build directed graph with 2 outgoing edges per node for musician
def build_musician_graph(musician_id):
    scale_notes = notes_by_musician[musician_id]
    nodes = []
    for _ in range(num_nodes):
        chord = make_chord(scale_notes)
        intensity = np.random.choice(intensities)
        duration = np.random.choice(durations)
        nodes.append((chord, intensity, duration))

    G = nx.DiGraph()
    for i, node in enumerate(nodes):
        G.add_node(i, data=node)

    for i in range(num_nodes):
        targets = set()
        while len(targets) < 2:
            tgt = np.random.randint(num_nodes)
            if tgt != i:
                targets.add(tgt)
        for tgt in targets:
            G.add_edge(i, tgt)

    # Transition matrix
    T = np.zeros((num_nodes, num_nodes))
    for i in range(num_nodes):
        neighbors = list(G.successors(i))
        p = 1.0 / len(neighbors)
        for n in neighbors:
            T[i, n] = p
```

```python
    return G, T

# Grover coin operator of dimension d
def grover_coin(d):
    return 2 * np.ones((d, d)) / d - np.eye(d)

# Build shift operator S on the space of positions x coin states
def build_shift_operator(G):
    n = G.number_of_nodes()
    max_deg = max(dict(G.out_degree()).values())
    size = n * max_deg

    S = np.zeros((size, size), dtype=complex)

    out_edges = {v: list(G.successors(v)) for v in G.nodes()}

    for v in G.nodes():
        for i_dir, w in enumerate(out_edges[v]):
            # Find direction at w that leads back to v (if exists)
            if v in out_edges[w]:
                j_dir = out_edges[w].index(v)
            else:
                j_dir = i_dir  # fallback if no reverse edge

            from_idx = v * max_deg + i_dir
            to_idx = w * max_deg + j_dir
            S[to_idx, from_idx] = 1
    return S

# Initialize uniform superposition over positions and coin states
def initial_state(num_nodes, coin_dim):
    psi = np.ones(num_nodes * coin_dim) / np.sqrt(num_nodes * coin_dim)
    return psi.astype(complex)

# Apply one coined quantum walk step
def coined_walk_step(psi, C, S, num_nodes, d):
    I = np.eye(num_nodes)
    coin_op = np.kron(I, C)
    U = S @ coin_op
    return U @ psi

# Measure position probabilities by summing over coin states
def measure_position(psi, num_nodes, d):
    probs = np.zeros(num_nodes)
    for v in range(num_nodes):
        probs[v] = np.sum(np.abs(psi[v*d:(v+1)*d]) ** 2)
```

```
    probs /= np.sum(probs)
    return probs

# Generate node sequence for musician using coined quantum walk
def generate_sequence_coined(G, steps=30, n_steps_per_note=7):
    n = G.number_of_nodes()
    d = max(dict(G.out_degree()).values())
    C = grover_coin(d)
    S = build_shift_operator(G)
    sequence = []
    current_pos = np.random.randint(n)  # Start at random node

    for _ in range(steps):
        # Initialize: walker at current_pos, uniform over coin states
        psi = np.zeros(n * d, dtype=complex)
        for c in range(d):
            psi[current_pos * d + c] = 1.0 / np.sqrt(d)
        # Quantum steps before measurement
        for _ in range(n_steps_per_note):
            psi = coined_walk_step(psi, C, S, n, d)
        # Measure
        probs = measure_position(psi, n, d)
        node = np.random.choice(range(n), p=probs)
        sequence.append(node)
        current_pos = node  # Next initial position
    return sequence


# Convert sequence to music21 stream with instrument
def create_music_stream(G, seq, instrument_name):
    stream = m21.stream.Stream()
    instr = getattr(m21.instrument, instrument_name)()
    stream.append(instr)
    velocity_map = {"pp": 40, "p": 55, "mp": 70, "mf": 85, "f": 100}

    for node in seq:
        chord_notes, intensity, duration = G.nodes[node]['data']
        chord_obj = m21.chord.Chord(chord_notes)
        chord_obj.duration = m21.duration.Duration(duration)
        chord_obj.volume.velocity = velocity_map[intensity]
        stream.append(chord_obj)
    return stream

# Main function for all 4 musicians
def generate_music_coined(num_steps=30):
    instruments = ["Piano", "Violin", "Flute", "Clarinet"]
    streams = []
```

```
    for musician_id in range(1, 5):
        G, T = build_musician_graph(musician_id)
        seq = generate_sequence_coined(G, steps=num_steps)
        stream = create_music_stream(G, seq, instruments[musician_id - 1])
        streams.append(stream)
    score = m21.stream.Score()
    for s in streams:
        score.append(s)
    return score

# Generate and save MIDI
score = generate_music_coined(num_steps=50)
score.write('midi', fp='My_Coined_music.mid')
print("Generated coined quantum walk MIDI: My_Coined_music.mid")
```

```
Generated coined quantum walk MIDI: My_Coined_music.mid
```

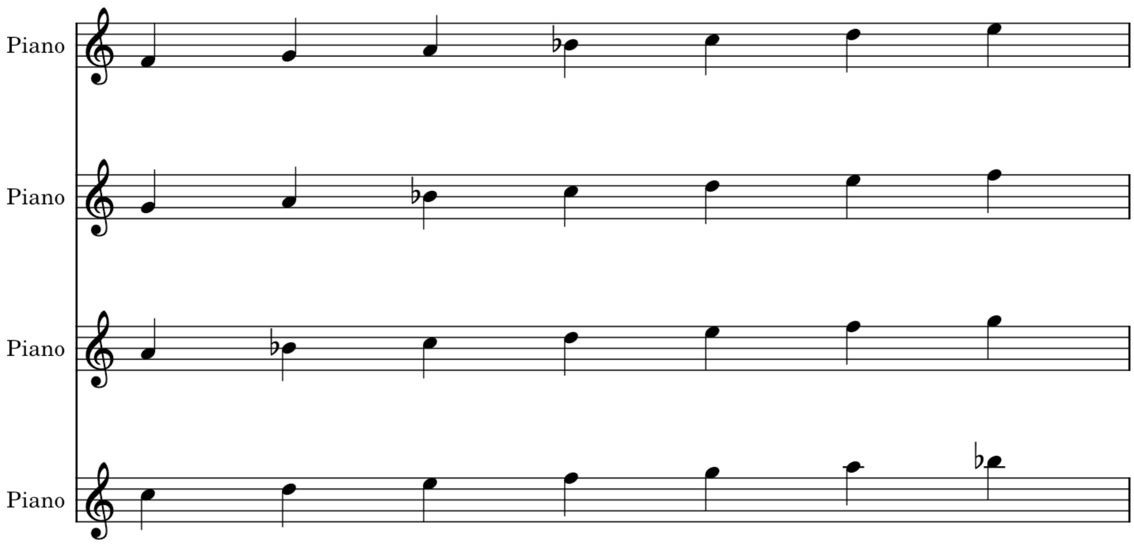
[ ]:

[ ]:

0.0.3 Scale diagram for all musicians

[7]:
```
import music21 as m21

# Define modes/scales for each musician
modes = [
    ("F Ionian (Major)",    ["F4", "G4", "A4", "Bb4", "C5", "D5", "E5"]),
    ("G Dorian",            ["G4", "A4", "Bb4", "C5", "D5", "E5", "F5"]),
    ("A Phrygian",          ["A4", "Bb4", "C5", "D5", "E5", "F5", "G5"]),
    ("C Mixolydian",        ["C5", "D5", "E5", "F5", "G5", "A5", "Bb5"]),
]

# Make one staff per mode
score = m21.stream.Score()
for name, notes in modes:
    part = m21.stream.Part()
    part.append(m21.clef.TrebleClef())
    part.append(m21.instrument.Piano())  # Just so MuseScore/MusicXML shows a␣
↪staff
    m = m21.stream.Measure()
    for n in notes:
        n_obj = m21.note.Note(n)
        n_obj.quarterLength = 1
        m.append(n_obj)
    part.append(m)
    part.append(m21.metadata.Metadata(title=name))
```

```python
    score.append(part)

# Save as MusicXML (import in MuseScore, print/export PDF)
score.write("musicxml", fp="quantum_modes_scales.musicxml")
#print("Staff for all four modes saved as: quantum_modes_scales.musicxml")

# Optional: Show score directly (opens in MuseScore if installed)
score.show()
```



[ ]:

[ ]: