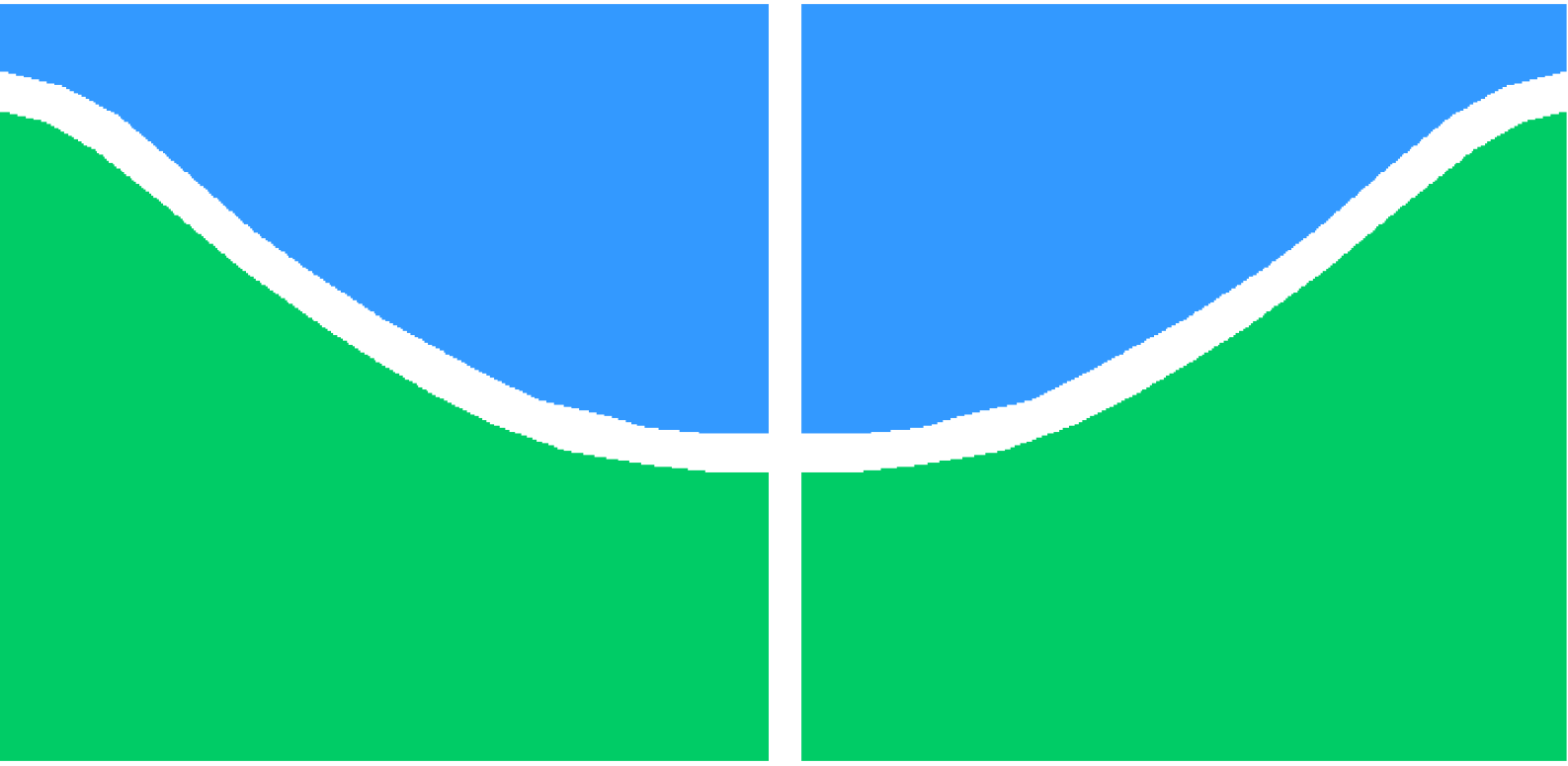

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Engenharia de *Software*

Desenvolvimento de um *software* de composição algorítmica de contrapontos palestrinianos

Autor: João Vitor Araujo Moura

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF

2018

João Vitor Araujo Moura

Desenvolvimento de um *software* de composição algorítmica de contrapontos palestrinianos

Monografia submetida ao curso de graduação em Engenharia de *Software* da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de *Software*.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF

2018

João Vitor Araujo Moura

Desenvolvimento de um *software* de composição algorítmica de contrapontos palestrinianos/ João Vitor Araujo Moura. – Brasília, DF, 2018-

65 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2018.

1. composição-algorítmica. 2. contrapontos. I. Prof. Dr. Edson Alves da Costa Júnior. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Desenvolvimento de um *software* de composição algorítmica de contrapontos palestrinianos

CDU 02:141:005.6

João Vitor Araujo Moura

Desenvolvimento de um *software* de composição algorítmica de contrapontos palestrinianos

Monografia submetida ao curso de graduação em Engenharia de *Software* da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de *Software*.

Brasília, DF, 09 de julho de 2018

Prof. Dr. Edson Alves da Costa Júnior
Orientador

Prof. Dr. Fernando William Cruz
Convidado 1

Prof. Dr. Henrique Gomes de Moura
Convidado 2

Brasília, DF
2018

Resumo

A composição algorítmica consiste na criação de peças musicais por meio de algoritmos computacionais, tendo como foco a automação e a imprevisibilidade, o uso desse tipo de algoritmo possui diversas aplicações. O objetivo desse trabalho é desenvolver um *software* capaz de compor algoritmicamente contrapontos palestrinianos para uma dada melodia. Esse desenvolvimento foi feito por meio de um estudo da teoria musical e das regras de contrapontos do século XVI, seguido por um levantamento de requisitos e pela construção de um protótipo com módulos de notação musical, intervalos, escalas e contrapontos capaz de gerar contrapontos de primeira espécie e servir de guia para o desenvolvimento dos módulos de contrapontos de segunda a quinta espécie. Inicialmente, utilizou-se de um algoritmo de busca completa, que foi trocado por uma solução que trata a geração de contrapontos como um grafo implícito, utilizando DFS combinada a programação dinâmica. O protótipo desenvolvido é capaz de ler nota em formato Lilypond e armazená-las, armazenar e utilizar intervalos e escalas, gerando contrapontos de primeira espécie.

Palavras-chaves: composição algorítmica, busca completa, grafos, dfs, dp, programação dinâmica, contrapontos palestrinianos.

Abstract

Algorithmic composition is the creation of music through computational algorithms, focusing on automation and unpredictability, usage of this type of algorithm has several applications. The goal of this project is to develop a software capable of composing counterpoint in the Palestrina style algorithmically to a given melody. This development was based on musical theory and Palestrina style counterpoint rules study, followed by requirements definition and development of a prototype with musical notation, intervals and scales modules that should be capable of generating first order counterpoints and serve as a guide to development of second to fifth counterpoint modules. Initially, a complete search algorithm was used and later changed to a solution that uses an implicit graph along with DFS and dynamic programming. The developed prototype is capable of reading notes on Lilypond format and store them, store and use intervals and scales, generating first order counterpoint.

Key-words: algorithmic composition, complete search, graphs, dfs, dp, dynamic programming, palestrina style counterpoints.

Lista de ilustrações

Lista de tabelas

Lista de Códigos

Lista de abreviaturas e siglas

DAG	*Direct Acyclic Graph*
BFS	*Breadth-First Search*
DFS	*Depth-First Search*
DP	*Dynamic Programming*
MIDI	*Musical Instrument Digital Interface*
XP	*Extreme Programming*

Lista de símbolos

♯	Acidente musical sustenido
♭	Acidente musical bemol
♮	Acidente musical bequadro

Sumário

Introdução

Desde sua criação, o computador é utilizado para automatizar tarefas antes realizadas por meio de esforço humano. Seja calculando ou montando partes de um automóvel, tais máquinas são capazes de realizar trabalhos de modo eficiente e com poucos erros. Contudo, uma barreira no uso de *softwares* e sistemas para automação de atividades encontra-se nas artes. Devido a seu caráter criativo e emocional, por muito tempo, imaginou-se ser impossível para um computador produzir arte – seja ela visual ou sonora. Mas é possível para uma máquina de calcular abstrair convenções de composição e compor peças musicais agradáveis aos ouvidos humanos, segundo Gogal e Goga (2004).

A área de pesquisa responsável pelo estudo da composição musical automatizada é conhecida como composição algorítmica. Ghose (2014) afirma que um *software* de composição algorítmica recebe parâmetros de entrada (como outras músicas, *inputs* de usuário ou dados randômicos) e, por meio de um algoritmo que possui convenções de composição como restrição, devolve uma melodia como saída.

Uma das áreas de composição musical passível de automatização é a composição de contrapontos musicais, definidos por Tragtenberg (1961) como melodias com qualidade harmônica coerente com a melodia principal. Sendo assim, este trabalho possui a seguinte questão de pesquisa: *Dada uma melodia monofônica, é possível gerar contrapontos palestrinianos algoritmicamente?*

Objetivos

Objetivo Geral

O objetivo geral deste trabalho é a implementação de um *software* de composição de contrapontos palestrinianos. A aplicação será implementada em C++, terá como entrada uma melodia principal por meio de formatos digitais de representação de música, como *Lilypond*[1] e *MusicXML*[2], e como saída uma melodia que serve como contraponto para a melodia dada, de acordo com as restrições definidas pelo usuário na interface do programa.

Objetivos Específicos

Os objetivos específicos deste trabalho são:

[1] <http://lilypond.org/>
[2] <https://www.musicxml.com/>

- implementar um algoritmo de leitura e interpretação de formatos digitais de representação de música;
- analisar os *softwares* de composição algorítmica de contrapontos já existentes;
- implementar um algoritmo de composição de contrapontos que respeite as restrições dos modos litúrgicos definidos.

Estrutura do Documento

Este documento divide-se em 4 capítulos. O capítulo 1 aborda a fundamentação teórica necessária para o projeto, trazendo conceitos de Engenharia de *Software* e da Teoria Musical. O capítulo 2 define a metodologia a ser utilizada, apresentando cronograma e a descrição específica das funcionalidades. O capítulo 3 apresenta os resultados obtidos: os algoritmos e o *software* desenvolvidos. O capítulo 4 traz as considerações finais, analisando o nível de sucesso do projeto e apresentando possibilidades de trabalhos futuros.

1 Fundamentação Teórica

Este capítulo apresenta a base teórica para o entendimento do algoritmo a ser implementado. Nele, são explicados conceitos referentes a teoria musical, grafos e paradigmas de solução de problemas. Na seção sobre teoria musical, são abordados os tópicos de notação musical, intervalos, escalas e contrapontos. Na seção sobre grafos, são explicadas classificações para um grafo, o que são grafos direcionados acíclicos e tipos de travessia em grafos. Na seção sobre paradigmas de solução de problemas, são abordados dois paradigmas de solução de problema: busca completa e programação dinâmica.

1.1 Teoria Musical

A música é a arte de combinar sons e silêncios. Segundo Med (1996), a música é constituída, principalmente, por melodia, harmonia, ritmo e contraponto. A melodia caracteriza-se por sons em ordem sucessiva, enquanto a harmonia caracteriza-se por sons em ordem simultânea. O ritmo é a ordem e proporção do sons de uma música. Já o contraponto é definido por meio de duas melodias dispostas simultaneamente.

Os sons são parte fundamental da música. Cada som possui diversas características como altura (frequência do som que o torna mais grave ou mais agudo), duração, intensidade e timbre. Quando o som é regular (possui altura definida), ele pode ser expresso por meio de notação musical.

1.1.1 Notação Musical

As notas musicais são utilizadas para representar os sons de uma música. Há sete notas musicais – usualmente nomeadas Dó, Ré, Mi, Fá, Sol, Lá, Si ou, respectivamente, C, D, E, F, G, A, B – que podem variar de acordo com a oitava ou acidentes musicais empregados, para representação de tais modificações, elas são dispostas em uma pauta, conforme exemplificado na Figura 1.

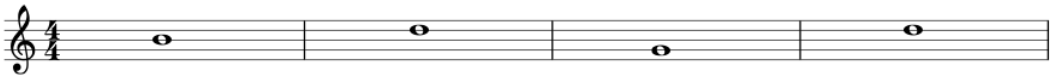

Figura 1 – Pauta musical

A pauta é utilizada para a representação de uma música, contendo linhas e espaços que definem a nota e em qual oitava está posicionada tendo como base a clave definida. Cada clave define uma nota específica e todas as outras notas da pauta são definidas a

partir desta. Por exemplo, a clave de Sol é a mais comumente usada e define que a nota da linha 2 é o G4 (Sol da quarta oitava, Figura 2).

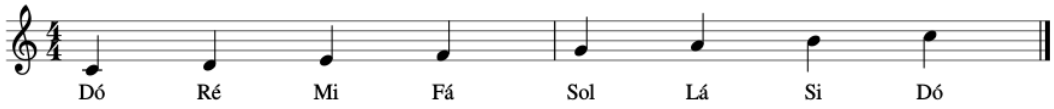

Figura 2 – Notas musicais na clave de Sol

Além da altura, cada som possui uma duração representada pela figura musical na pauta. No sistema atual, começa-se pela semibreve, que possui valor 1. A partir dela, a próxima subdivisão equivale à metade do tempo da anterior. Sendo assim, a mínima possui metade da duração da semibreve e possui valor 2, a semínima possui metade da duração da mínima e possui valor 4 e assim sucessivamente. As figuras, seus valores e durações estão representados na Tabela 1.

Tabela 1 – Figuras Musicais

Nome	**Figura**	**Valor**	**Duração**
Semibreve	𝅝	1	2 𝅗𝅥
Mínima	𝅗𝅥	2	1/2 𝅝 ou 2 ♩
Semínima	♩	4	1/2 𝅗𝅥 ou 2 ♪
Colcheia	♪	8	1/2 ♩ ou 2 𝅘𝅥𝅯
Semicolcheia	𝅘𝅥𝅯	16	1/2 ♪

A duração em segundos de uma nota é definida pela sua figura e pelo andamento da música. O andamento define quantas notas de uma determinada figura de ritmo pode ser tocada em um espaço de tempo. Comumente, utiliza-se a quantidade de semínimas que podem ser tocadas durante um minuto para se definir o andamento de uma música.

Em uma pauta, as figuras de ritmo são agrupadas em compassos. Cada compasso pode agrupar um determinado número de notas de acordo com sua estrutura, definida pela quantidade de figuras e a figura de ritmo usada como base. Por exemplo, um compasso 3/4 possui 3 como a quantidade de valores e 4 como a figura de ritmo, sendo assim, esse tipo de compasso é preenchido por três semínimas – que possui valor 4. Desse modo, é possível também encaixar seis colcheias (que possui metade da duração de uma semínima) ou uma mínima e uma semínima em um compasso 3/4. Na música, a estrutura do compasso também define os tempos fortes e fracos da música. Em uma música 2/4, por exemplo, o primeiro tempo é forte (ou *arsis*) e o segundo tempo é fraco (ou *thesis*). Veja a Figura 3.

Outro elemento que pode ser representado na pauta são os acidentes musicais. Cada nota possui a distância relativa de um tom ou um semitom das notas adjacentes, como representado na Figura 4. Essa distância calculada em tons e semitons também

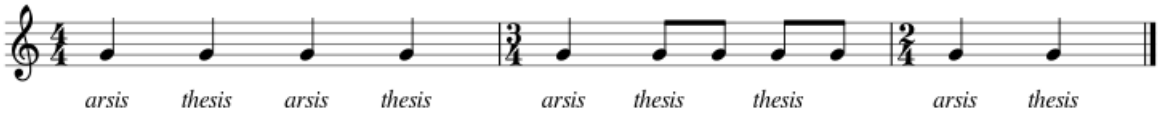

Figura 3 – Exemplos de compasso e seus respectivos tempos fortes e fracos

é utilizada para a definição de intervalos e pode ser modificada por meio de acidentes musicais.

Existem dois tipos de acidentes musicais, o bemol (♭) e o sustenido (♯). O bemol abaixa a nota original em um semitom, como exemplo, o Fá e o Sol possuem um tom entre eles, porém, o Fá e o Sol bemol possuem um semitom entre eles. Já o sustenido aumenta a nota original em um semitom. Outras variações desses acidentes são o dobrado bemol, que abaixa a nota em um tom, o dobrado sustenido, que aumenta a nota em um tom e o bequadro, que retorna uma nota modificada ao seu valor original. Os símbolos de cada acidente estão representados na Figura 4.

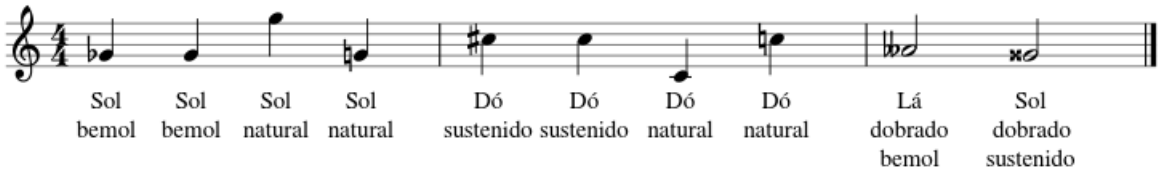

Figura 4 – Acidentes Musicais

Esses acidentes podem ser pontuais ou recorrentes durante uma música. Quando são pontuais, eles são expressos ao lado da nota que modificam, modificando também as notas de mesma altura naquele compasso. Quando são recorrentes, são representados no início da clave ou do primeiro compasso que modificam. Enquanto a representação pontual modifica apenas um único som em uma oitava específica, a representação no início modifica qualquer som daquela mesma nota, e a esse conjunto de acidentes representados no início da pauta dá-se o nome de armadura de clave. Veja a Figura 5.

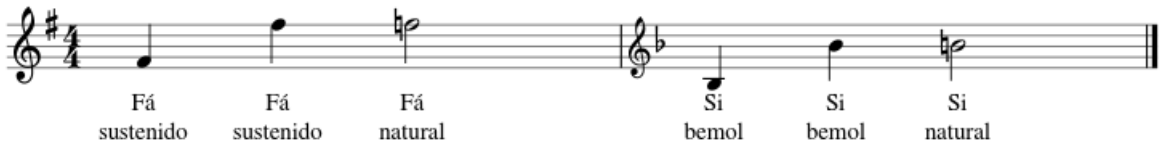

Figura 5 – Armaduras de Clave

1.1.2 Intervalos

A diferença de altura entre duas notas pode ser classificada por meio de intervalos. Os intervalos podem ser harmônicos (quando se compara duas notas simultâneas) e melódicos (quando se compara duas notas em sequência). Um intervalo melódico pode ser ascendente, se a primeira nota for mais grave que a segunda, e descendente, se a primeira nota for mais aguda que a segunda.

Os intervalos também são classificados quantitativamente e qualitativamente. A classificação quantitativa de um intervalo tem como base a quantidade de notas entre as duas notas, incluindo-as e ignorando acidentes musicais. Por exemplo, a classificação quantitativa do intervalo entre o G4 (Sol na quarta oitava) e o E5 (Mi na quinta oitava) é sexta (6ª), que é a mesma que o intervalo entre G♯4 (Sol sustenido na quarta oitava) e E♭5 (Mi bemol na quinta oitava). Já a classificação qualitativa tem como base o número de tons e semitons entre as duas notas analisadas. Dependendo da quantidade de semitons, o intervalo pode ser justo (no caso de intervalos de primeira, quarta, quinta e oitava), maior, menor (no caso de intervalos de segunda, terça, sexta e sétima), aumentados ou diminutos.

Cada intervalo pode ser classificado como consonante ou dissonante, uma classificação que define se o som provocado pelas duas notas soando seguidas ou em paralelo causam um efeito de repouso ou tensão, respectivamente. A classificação de um intervalo em relação ao seu efeito pode variar de acordo com a época ou estilo musical. Neste trabalho, será adotada a classificação correspondente às regras do contraponto modal do século XVI definidas por Jeppesen (1992).

1.1.2.1 Intervalos Justos

Os intervalos de primeira, quarta, quinta e oitava podem ser classificados como justos, dependendo da distância em semitons.

Uma nota é a primeira justa (1ªJ ou P1[1]) de outra se elas tiverem a mesma altura, ou seja, apenas uma nota entre elas e nenhum semitom de diferença, esse intervalo também é chamado de uníssono (Figura 6).

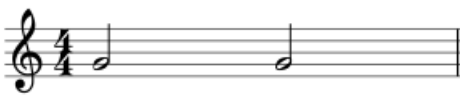

Figura 6 – Exemplo de Primeira Justa

A quarta justa (4ªJ ou P4) ocorre quando há quatro notas entre elas e a diferença é de dois tons e um semitom. Veja a Figura 7.

[1] Sigla para *perfect first*

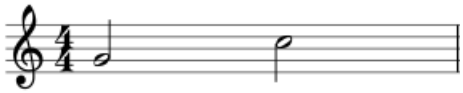

Figura 7 – Exemplo de Quarta Justa

A quinta justa (5ªJ ou P5) ocorre quando há cinco notas entre elas e a diferença é de três tons e um semitom. A Figura 8 traz um exemplo de quinta justa.

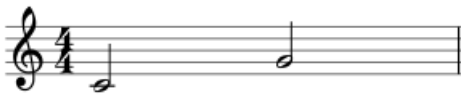

Figura 8 – Exemplo de Quinta Justa

A oitava justa (8ªJ ou P8) ocorre quando há oito notas entre elas e a diferença é de cinco tons e dois semitons (Figura 9).

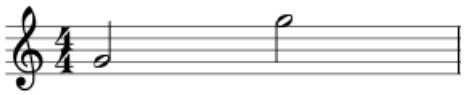

Figura 9 – Exemplo de Oitava Justa

Dentre os intervalos justos, todos, exceto a quarta justa, são considerados consonâncias perfeitas. Vale ressaltar que, segundo Med (1996), a quarta justa é considerada um intervalo consonante na música contemporânea.

1.1.2.2 Intervalos Maiores e Menores

Os intervalos de segunda, terça, sexta e sétima podem ser classificados como maiores ou menores, dependendo da distância em semitons.

A segunda ocorre quando há duas notas entre elas. A segunda menor (2ªm ou m2) ocorre quando a diferença é de um semitom, já a segunda maior (2ªM ou M2) possui uma diferença de um tom. Veja a Figura 10.

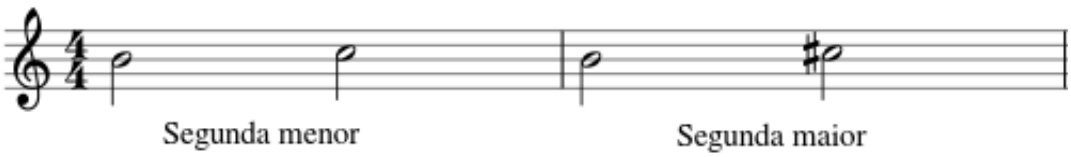

Figura 10 – Exemplos de Segunda Menor e Segunda Maior

A terça ocorre quando há três notas entre elas. A terça menor (3ªm ou m3) ocorre quando a diferença é de um tom e um semitom, já a terça maior (3ªM ou M3) possui uma diferença de dois tons (Figura 11).

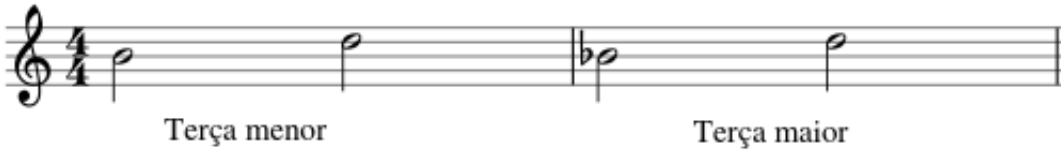

Figura 11 – Exemplos de Terça Menor e Terça Maior

A sexta ocorre quando há seis notas entre elas. A sexta menor (6ªm ou m6) ocorre quando a diferença é de três tons e dois semitons, já a sexta maior (6ªM ou M6) possui uma diferença de quatro tons e um semitom. Veja a Figura 12.

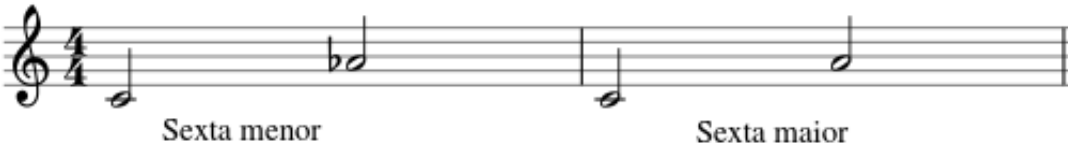

Figura 12 – Exemplos de Sexta Menor e Sexta Maior

A sétima ocorre quando há sete notas entre elas. A sétima menor (7ªm ou m7) ocorre quando a diferença é de quatro tons e dois semitons, já a sétima maior (7ªM ou M7) possui uma diferença de cinco tons e um semitom (Figura 13).

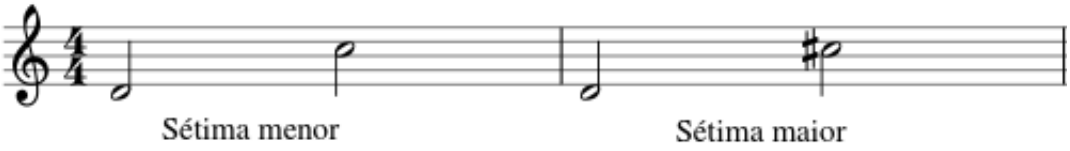

Figura 13 – Exemplos de Sétima Menor e Sétima Maior

Os intervalos de terça maior e menor e sexta maior e menor são classificados como consonâncias imperfeitas, enquanto os intervalos de segunda maior e menor e sétima maior e menor são classificados como dissonâncias.

1.1.2.3 Intervalos Aumentados e Diminutos

Se a distância em semitons das duas notas do intervalo não identificá-lo como justo, maior ou menor, ele será classificado como aumentado ou diminuto.

Intervalos aumentados possuem mais semitons que os intervalos justos ou maiores. Se o intervalo possuir um semitom a mais, ele é chamado de aumentado, se forem dois semitons a mais, é chamado de superaumentado, se forem três semitons a mais, é chamado de três vezes aumentado e assim sucessivamente, até os maiores intervalos possíveis que

são os cinco vezes aumentados. Na Figura 14, há exemplos de quarta aumentada (4ªA ou A4), quarta superaumentada (4ªSA ou SA4[2]) e quarta 3x aumentada (4ª3xA ou 3xA4[3]).

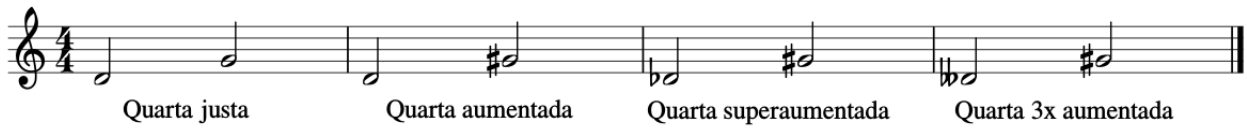

Figura 14 – Exemplos de Intervalos Aumentados

Intervalos diminutos possuem menos semitons que os intervalos justos ou menores. Se o intervalo possuir um semitom a menos, ele é chamado de diminutos, se forem dois semitons a menos, é chamado de superdiminuto, se forem três semitons a menos, é chamado de três vezes diminuto e assim sucessivamente, até os menores intervalos possíveis que são os cinco vezes diminutos. Na Figura 15, há exemplos de sexta diminuta (6ªd ou d6), sexta superaumentada (6ªsd ou sd6[4]) e sexta 3x aumentada (6ª3xd ou 3xd6[5]).

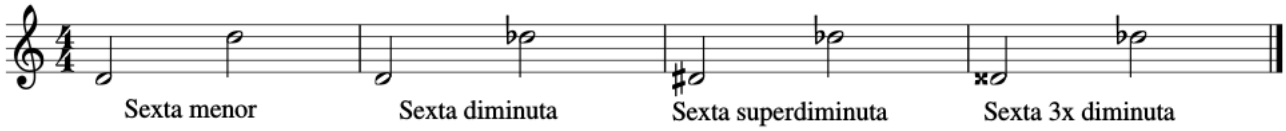

Figura 15 – Exemplos de Intervalos Diminutos

Os intervalos aumentados e diminutos são classificados como consonantes e dissonantes de acordo com os intervalos justos, maiores e menores que possuem a mesma distância em semitons, por exemplo, o intervalo de quarta superaumentada é considerado consonante pois possui a mesma distância em semitons que a quinta justa, embora sua classificação quantitativa seja a mesma que a quarta justa, classificada como dissonância. Por causa disso, os intervalos aumentados e diminutos são considerados dissonâncias condicionais.

1.1.2.4 Intervalos Compostos

Se um intervalo ultrapassar a oitava justa, quantitativa ou qualitativamente, ele é considerado composto. Intervalos compostos tem sua classificação definida de acordo com seus correspondentes simples. Para classificar um intervalo composto, divide-se o valor quantitativo por 7 e o resto é utilizado para definir seu equivalente simples, por exemplo, uma 15ª corresponde a uma 1ª.

Para a análise quantitativa, divide-se o valor de semitons por 12 e o resto é utilizado para definir se ele é justo, maior, menor, diminuto ou aumentado, por exemplo uma 10ª

[2] Notação utilizada no código para intervalos superaumentados
[3] Notação utilizada no código para intervalos 3x, 4x e 5x aumentados
[4] Notação utilizada no código para intervalos superdiminutos
[5] Notação utilizada no código para intervalos 3x, 4x e 5x diminutos

com uma diferença de 16 semitons é equivalente a uma 3ª com 4 semitons, sendo uma 3ª maior, logo, o intervalo analisado é uma 10ª maior.

Os intervalos compostos são, basicamente, seus equivalentes simples somados a oitavas justas, como exemplificado na Figura 16. O intervalo de oitava aumentada é considerado composto e é equivalente ao intervalo de primeira aumentada.

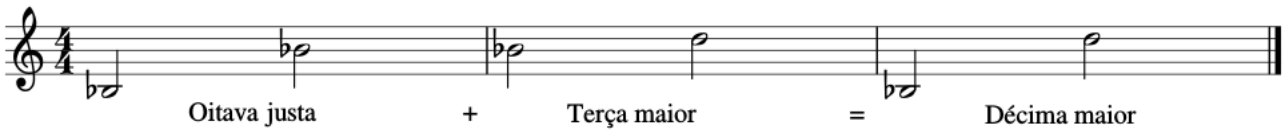

Figura 16 – Exemplo de Intervalo Composto

1.1.3 Escalas

Segundo Med, escala é o conjunto de notas disponíveis em um sistema musical. As notas presentes em uma escala possuem intervalos específicos entre si. As escalas podem ser classificadas quanto ao número de notas, sendo as mais conhecidas a pentatônica (cinco notas), a hexacordal (seis notas), a heptatônica (sete notas) e a cromática (doze notas). A escala cromática, por exemplo, possui todas as dozes notas (com e sem acidentes) de uma oitava, tendo intervalos de segunda menor ou primeira aumentada entre elas (Figura 17).

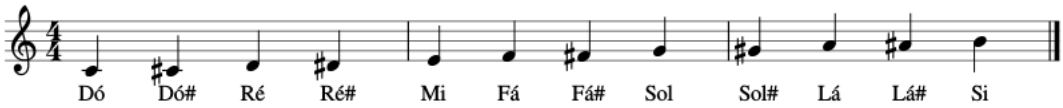

Figura 17 – Escala Cromática

Outra classificação é quanto à utilização, as mais conhecidas são as escalas classificadas como diatônicas, que possuem oito notas, sete distintas e a última sendo a primeira uma oitava acima, possuindo intervalos de segunda maior ou menor entre elas. Em uma escala diatônica, pode-se definir quais notas fazem parte dela observando a armadura de clave presente na partitura. Por exemplo, na escala representada na figura 18, há uma armadura de sustenindo em F (Fá) e em C (Dó), logo, a escala possui as seguintes notas: D, E, F♯, G, A, B e C♯. Músicas compostas nessa escala não podem possuir notas como A♭, B♯ ou C, a não ser em caso de acidentes pontuais, definidos pelo compositor. Cada nota de uma escala diatônica possui um grau (que varia de um a sete) e um nome específicos. A principal nota é a de grau I, chamada tônica. Ela dá nome à escala e a inicia, tendo todas as outras notas definidas a partir dela.

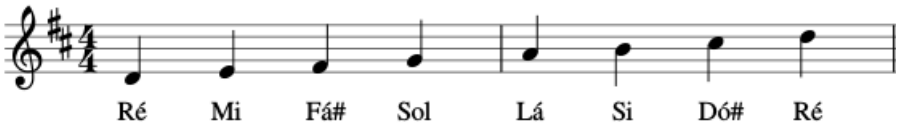

Figura 18 – Exemplo de Escala Diatônica

1.1.3.1 Escalas Maiores

As escalas diatõnicas podem ser classificadas como maiores ou menores. As escalas maiores iniciam na tônica e possuem os seguintes intervalos, em tons, entre notas consecutivas: tom - tom - semitom - tom - tom - tom - semitom. Classificando os intervalos, são: 2ªM - 2ªM - 2ªm - 2ªM - 2ªM - 2ªM - 2ªm. Nas Figuras 19 e 20, estão representadas as escalas de Dó maior (que não possui nenhum acidente musical em sua armadura) e a de Fá maior (que possui um bemol em Si em sua armadura), respectivamente.

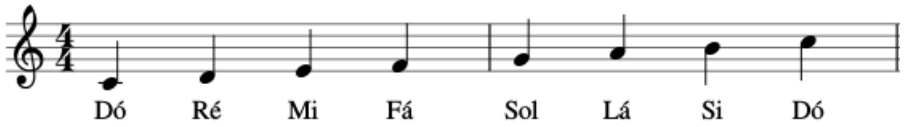

Figura 19 – Escala de Dó Maior

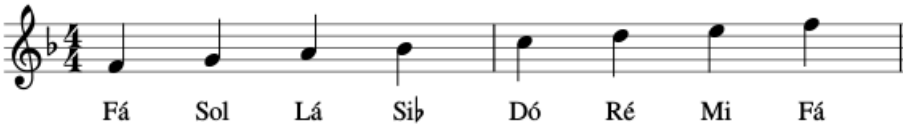

Figura 20 – Escala de Fá Maior

1.1.3.2 Escalas Menores

Assim como as escalas maiores, as escalas menores iniciam na tônica e possuem os seguintes intervalos, em tons, entre notas consecutivas: tom - semitom - tom - tom - semitom - um tom e meio - semitom. Classificando os intervalos, são: 2ªM - 2ªM - 2ªm - 2ªM - 2ªM - 2ªM - 2ªm. Nas Figuras 21 e 22, estão representadas as escalas de Lá menor (que não possui nenhum acidente musical em sua armadura) e a de Mi menor (que possui um sustenido em Fá em sua armadura).

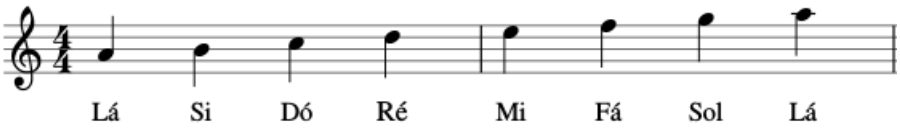

Figura 21 – Escala de Lá Menor

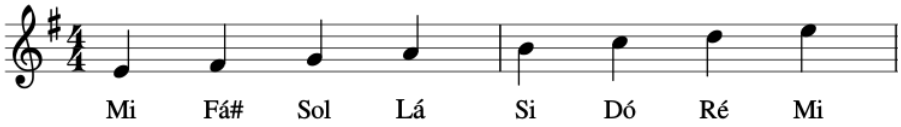

Figura 22 – Escala de Mi Menor

As escalas são utilizadas na composição de contrapontos definindo quais notas podem ser utilizadas para a geração do contraponto, de acordo com a escala da melodia principal.

1.1.4 Contraponto

Koellreuter e Joachim (1996) afirmam que contraponto é a arte de tornar independentes linhas melódicas de expressão autônoma, ou seja, combinar duas linhas melódicas simultâneas de forma que sejam melódicamente independentes, mas harmonicamente interdependentes. O objeto de estudo desse trabalho é o contraponto modal do século XVI, também chamado de contraponto palestriniano. Esse tipo de contraponto possui diversas espécies, cada uma definida pelo número de notas que um contraponto deve ter para cada nota da melodia principal (ou *cantus firmus*). Nesse trabalho, serão tratados os contrapontos de primeira e segunda espécie (Figuras 23 e 24).

1.1.4.1 Contraponto de Primeira Espécie

Segundo Jeppesen (1992), o contraponto de primeira espécie possui as seguintes regras:

1. deve haver uma nota no contraponto para cada nota do *cantus firmus*;
2. os intervalos harmônicos entre o contraponto e o *cantus firmus* devem ser consonates;
3. deve-se começar e terminar em consonâncias perfeitas;
4. se o contraponto for inferior (suas notas são mais graves que a do *cantus firmus*), somente a oitava justa e o uníssono podem ser usados no início e no final;
5. o uníssono só pode ocorrer no início ou no final;
6. não é permitido oitavas e quintas paralelas (dois ou mais intervalos de oitavas ou de quintas seguidos);
7. os intervalos não podem ser maiores que uma décima maior (10^{a}M);
8. só são permitidos, no máximo, quatro intervalos seguidos de terças ou sextas (maiores ou menores) seguidos;

9. se o movimento for paralelo, os intervalos melódicos devem ser menores que uma quarta ou um salto de oitava;

10. deve-se priorizar movimento contrário (utilizar intervalo melódico ascendente se o *cantus firmus* estiver em intervalo melódico descendente e vice-versa).

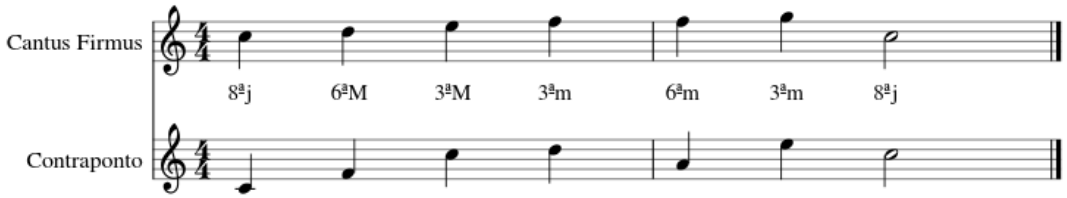

Figura 23 – Exemplo de Contraponto de Primeira Espécie

1.1.4.2 Contraponto de Segunda Espécie

Segundo Jeppesen (1992), o contraponto de segunda espécie possui, além das regras do contraponto de primeira espécie, as seguintes regras:

1. dissonâncias podem ser utilizadas em *arsis* (porções não-acentuadas do compasso);

2. as dissonâncias podem ser utilizadas apenas como nota de passagem, entre duas consonâncias e formando um grau conjunto, isto é, o intervalo melódico; entre a nota anterior e posterior à nota que provoca dissonância deve ser de segunda, maior ou menor;

3. o uníssono só pode ser utilizado no início, final ou em *arsis*;

4. se o uníssono for alcançado por salto (intervalo maior que uma segunda), deve ser deixado em grau conjunto na direção oposta à do salto;

5. evitar quintas e oitavas em *thesis* sucessivas, por se aproximarem de quintas e oitavas paralelas.

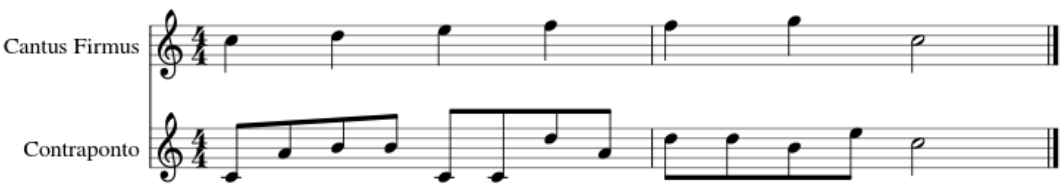

Figura 24 – Exemplo de Contraponto de Segunda Espécie

1.2 Grafos

Na geração de um contraponto, cada nota dele depende da nota dos *cantus firmus* tocada em simultâneo devido a regras definidas com base no intervalo harmônico – como a proibição de intervalos dissonantes, por exemplo. Porém, as notas disponíveis para um dado tempo do contraponto também depende das notas anteriores a ela devido a regras relacionadas à intervalos melódicos e repetição ou uso demasiado de intervalos harmônicos repetidos – como a proibição de oitavas paralelas. Sendo assim, a nota atual define, em conjunto com a próxima nota do *cantus firmus*, quais notas podem ser empregadas no próximo tempo. Dada tal organização, essa estrutura pode ser representada por um grafo.

Grafo é uma estrutura formada por vértices (os nós que representam a unidade fundamental da estrutura) e arestas (ligações que indicam quais outros vértices podem ser alcançados a partir de um). Na Figura 25 está representado um exemplo de estrutura formada pelas notas possíveis em um contraponto. Vale ressaltar que, segundo Halim (2013), como a estrutura do contraponto não é diretamente representada por um grafo na solução proposta, ela é definida como um grafo implícito.

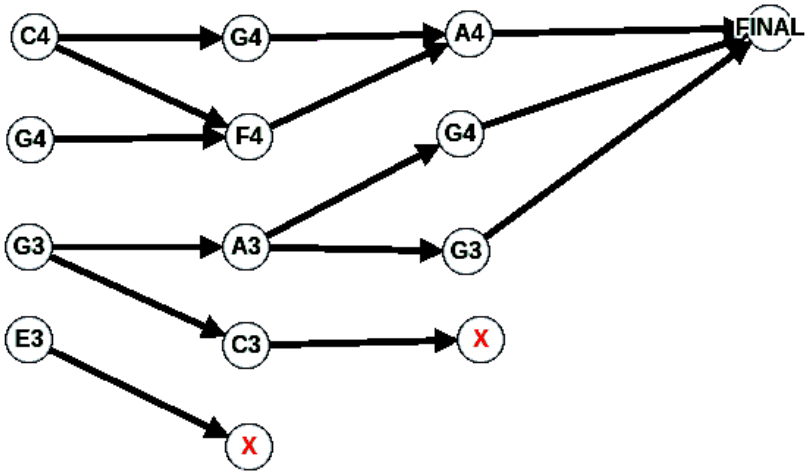

Figura 25 – Exemplo de Notas Possíveis Representados em Grafo

1.2.1 Grafos Direcionados Acíclicos

Um grafo pode possui diversas classificações. Uma classificação clássica é quanto à sua direção. Se as arestas de um grafo não definem a direção da ligação, isto é, se os vértices X e Y estão ligados por uma aresta, significa que X é diretamente alcançável (apenas um vértice é necessário para navegar da origem ao destino) a partir de Y e Y é diretamente alcançável a partir de X, esse grafo é classificado como não-direcionado. Mas se as arestas possuem direção e X ser diretamente alcançável por Y não garante que Y é diretamente alcançável a partir de X, esse grafo é direcionado, como afirma Halim (2013). Veja a Figura 26.

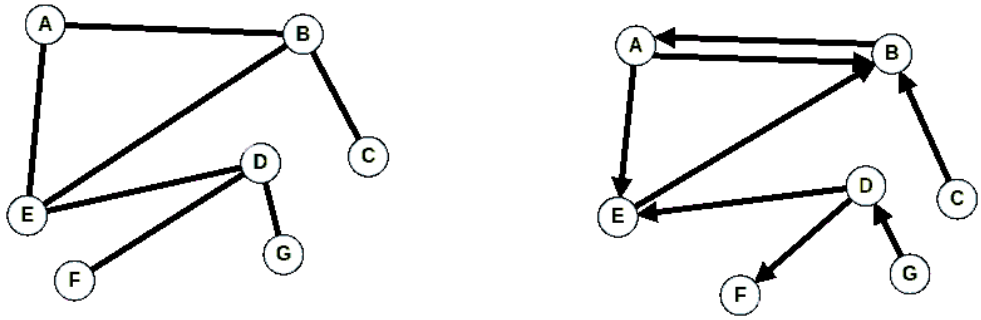

Figura 26 – Exemplos de Grafo Não-Direcionado e Direcionado

Grafos direcionados podem ser classificados como cíclicos ou acíclicos. Grafos acíclicos são grafos que não possuem ciclos, isto é, para todo vértice A, se A alcança B, B não alcança A. Se para pelo menos um par de vértices A e B, A alcançar B e B alcançar A, esse grafo é considerado cíclico.

Dadas tais definições, é possível perceber que a estrutura do contraponto é um grafo direcionado acíclico (ou DAG, sigla para *Direct Acyclic Graph*). Nesse tipo de grafo, cada vértice possui um ou mais pais – vértices que apontam para ele. Assemelhando-se à estrutura de uma árvore (grafo com raiz definida como o único vértice sem pai e que cada outro vértice tem apenas um pai), o DAG difere-se por cada vértice por ter mais de um pai e vários vértices sem pai, mas mantém a propriedade de ser navegável em apenas uma direção, impedindo a formação de *loops* que tornariam a exploração da solução um processo infinito. A Figura 27 exemplifica esses três tipos de grafos.

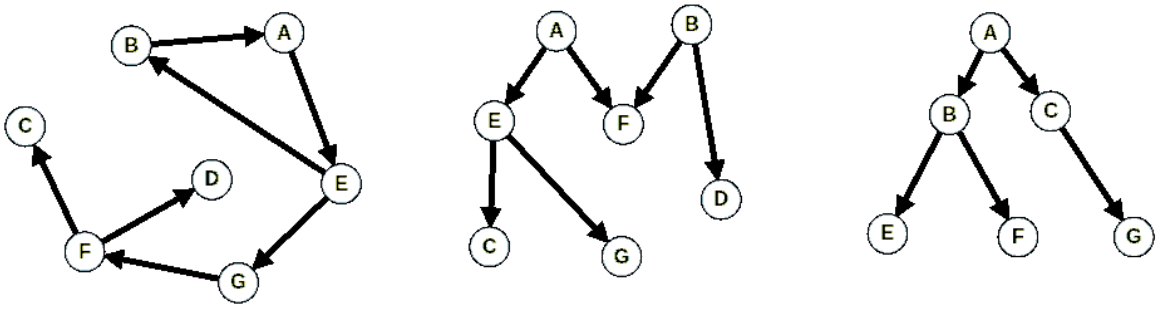

Figura 27 – Exemplos de Grafo Não-DAG, DAG e Árvore

1.2.2 Travessia de Grafos

Há diversas formas de se visitar todos os vértices de um grafo. O número total de formas é igual ao fatorial do número de vértices, pois uma travessia difere-se de outra apenas pela ordem em que os vértices são visitados. Contudo, algumas travessias possuem propriedades específicas e utilizam as arestas para definí-las. As duas travessias mais

conhecidas são a busca por profundidade (DFS, sigla para *Depth-First Search*) e a busca por largura (BFS, sigla para *Breadth-First Search*).

A busca por profundidade funciona de forma recursiva. Primeiramente, um vértice é escolhido como o vértice inicial, antes de explorá-lo (analisar o valor contido no vértice), visitamos todos as suas conexões, para cada vértice conectado, antes de explorá-lo, visitamos sua conexões. Esse processo de visita ocorre até atingir um vértice que não possui arestas ou possui apenas conexões já visitadas. Ao chegar nesse ponto, o algoritmo explora, recursivamente, os vértices previamente visitados. Utilizando uma estrutura de dados conhecida, uma DFS pode ser implementada por meio do uso de pilhas, devido à propriedade delas que definem que o último a entrar na estrutura é o primeiro a sair. Veja a Figura 28.

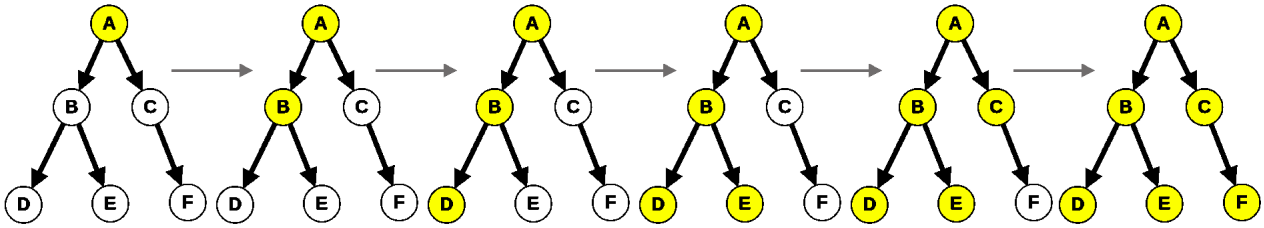

Figura 28 – Exemplo de DFS

A busca por largura inicia em um vértice e visita todos os filhos (vértices para os quais o vértice atual aponta) antes de visitar os filhos destes. Desse modo, em um DAG, o primeiro nível (composto por vértices sem pai) é completamente visitado antes de o segundo nível (filhos destes) ser visitado e assim sucessivamente. Utilizando uma estrutura de dados conhecida, uma BFS pode ser implementado por meio do uso de filas devida à propriedade delas que define que o primeiro a entrar na estrutura é o primeiro a sair. Veja a Figura 29.

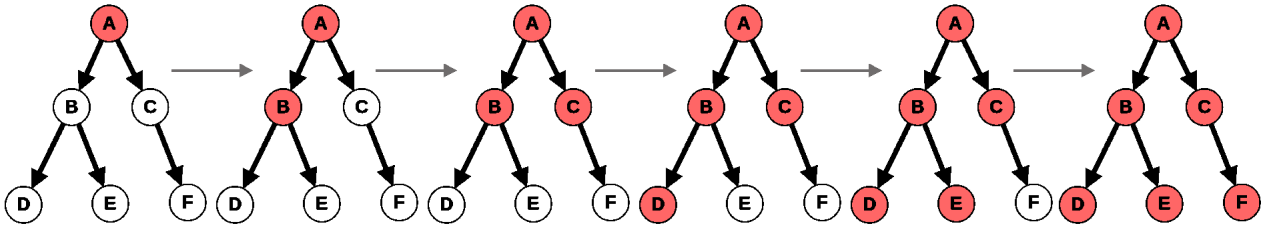

Figura 29 – Exemplo de BFS

Para a solução proposta nesse trabalho será usada a DFS por dois motivos. O primeiro é que a DFS chega a uma folha (vértice sem filhos) primeiro, o que é interessante para a solução pois busca-se a primeira solução disponível. O segundo motivo é que a DFS permite que seja utilizado um único vetor, inserindo e retirando no final deste (consequentemente, funcionando de forma análoga a uma pilha), economizando memória

e diminuindo o tempo de execução que levaria para fazer cópias do vetor em uma BFS, pois o comportamento análogo a uma fila exigiria isso.

1.3 Paradigmas de Solução de Problema

O objetivo de encontrar uma sequência de notas que se encaixe nas regras do contraponto pode ser alcançado pela análise de todas as combinações de notas possíveis, verificando se cada uma delas atende às regras e retornando como resposta a primeira sequência que atender. A esse tipo de busca pela solução se dá o nome de Busca Completa.

1.3.1 Busca Completa

Halim (2013) afirma que na Busca Completa, todo ou parte do espaço de soluções possíveis é explorado. No código 1.1 é possível observar a geração de todas as combinações de notas possíveis para um contraponto por meio de um algoritmo de *backtracking* (algoritmo recursivo de busca completa). Contudo, nesse caso, tal abordagem possui alta complexidade em termos de execução. Suponhando um espaço amostral com 88 notas disponíveis e um *cantus firmus* com 50 notas, apenas para o contraponto de primeira espécie há 88^{50} sequências possíveis.

Código 1.1 – Busca Completa

```
// Vector with 88 musicais notes found on piano
vector<Note> available_notes(88);

void backtracking(vector<Note> counterpoint) {
  // Process solution if
  // counterpoint candidate and cantus firmus
  // has the same number of notes
  if(counterpoint.size() == CANTUS_FIRMUS_SIZE){
    process_solution(counterpoint);
  }

  // Append each available note to counterpoint
  for(int i = 0; i < (int) available_notes.size(); i++){
    counterpoint.push_back(available_notes[i]);
    backtracking(counterpoint);
    counterpoint.pop_back();
  }
}
```

Para diminuir a complexidade de uma busca completa, podas podem ser aplicadas. Podas são aplicadas quando uma parte gerada da solução já indica que ela certamente não gerará uma solução. No caso do contraponto, por exemplo, podemos podar soluções parciais que já apresentem um número de movimentos reversos maior que o definido pelas regras mesmo antes de chegarem à última nota.

1.3.2 Programação Dinâmica

Mesmo após a aplicações de todas as podas possíveis, ainda há o caso de diversos estados serem recalculados. Como explicado anteriormente, uma nota em nosso grafo pode ter mais de um pai, isto é, mais de uma nota pode ter a nota atual como a seguinte em uma possível solução. Entretanto, uma vez alcançada a nota atual, o resto da solução é o mesmo, sendo redundante calcular esse estado (e se ele levará a possíveis soluções ou não) mais de uma vez. Tendo estados que se repetem entre soluções, é possível aplicar um paradigma de solução de problemas chamado Programação Dinâmica.

A Programação Dinâmica (ou DP, sigla para *Dynamic Programming*) é um paradigma de solução de problema baseado em estados, transições e casos-base, segundo Halim (2013). Cada estado resolve um problema e para isso ele precisa resolver subproblemas. A forma como o estado atual se relaciona com seus subestados é denominada transição.

Dada a forma recursiva como a DP funciona, casos-base garantem que o algoritmo não assuma uma forma infinita. Esses casos são aqueles com solução trivial que não necessita da resolução de subproblemas para respondê-la. A ideia central da DP é que subproblemas se repetem durante a solução, portanto, um estado já calculado deve ter seu resultado armazenado para futuras consultas que provavelmente acontecerão durante a resolução do problema.

Um exemplo clássico de como uma DP pode agilizar a resolução de um problema com subestados repetidos é o cálculo da sequência de Fibonacci. Os números da sequência de Fibonacci seguem a seguinte regra:

$$F(n) = \begin{cases} n & \text{se } n \leqslant 1 \\ F(n-1) + F(n-2) & \text{c.c.} \end{cases}$$

No cálculo dos números de Fibonacci, o estado em uma dada posição N é o valor do N-ésimo número de Fibonacci. Para calculá-lo, utiliza-se os dois números de Fibonacci anteriores a ele na sequência, sendo essa a transição do algoritmo. Os únicos dois casos em que o cálculo não necessita dos dois anteriores são os números de Fibonacci nas posições 0 e 1, para esses, a solução é, por definição, 0 e 1, respectivamente. Dessa forma, 0 e 1 são nossos casos-base.

Tomando como exemplo o cálculo de F(5), calculando-o sem memorização de estados, é necessário calcular F(4) e F(3). Para calcular o F(4), é preciso calcular o F(3) e o F(2). Sem a memorização, após o cálculo do F(4), o F(3), que já havia sido calculado para o F(4), será recalculado, bem como o F(2) para o cálculo do F(3). Contudo, se o valor de F(3) for memorizado em alguma estrutura de dados (geralmente um vetor) para consultas futuras, seria necessário calcular cada estado apenas uma vez. Na figura 30 há um comparativo entre a árvore de chamadas para F(5) com e sem a memorização característica de uma DP, os nós em amarelo representam os nós que utilizam recursão, os nós vermelhos são os casos-base e os nós verdes utilizam valores previamente memorizados.

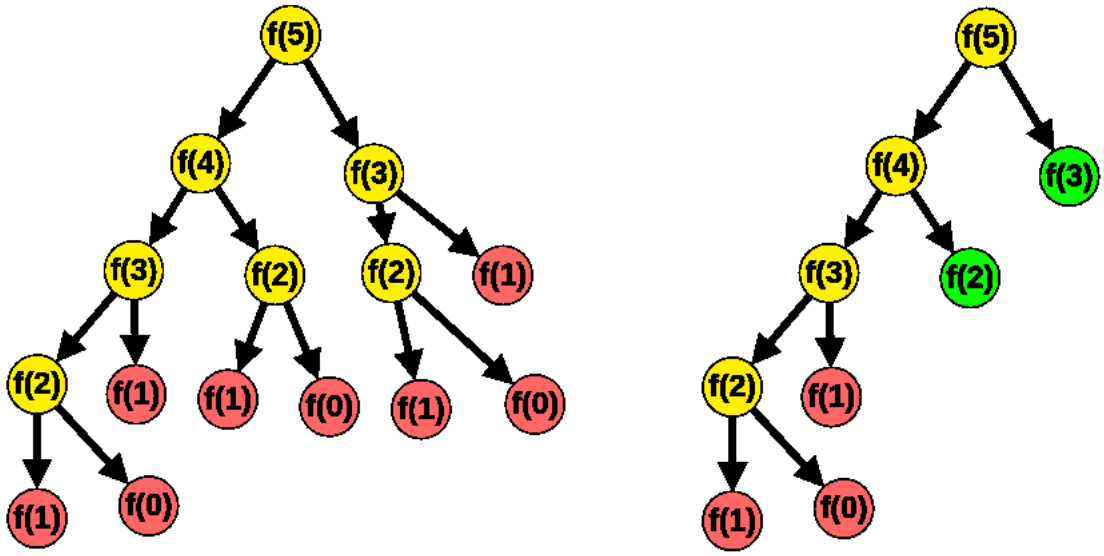

Figura 30 – Comparativo entre a abordagem não-DP e DP para o cálculo do F(5)

Neste trabalho, a DP será utilizada para a geração do contraponto. O estado sendo caracterizado por uma dada nota em uma dada posição no contraponto com um número específico de movimentos paralelos e terças e sextas paralelas. A transição é definida pela solução da próxima nota no contraponto, partindo da nota atual. O caso-base é definido por quando o algoritmo atinge o final do *cantus firmus*, retornando que aquela solução é válida, já que não foi podada nenhuma vez durante a construção daquela solução.

2 Metodologia

A implementação do *software* utilizará a metodologia ágil *Extreme Programming* (XP) como ciclo de vida de desenvolvimento, dado o escopo, o nível de definição dos requisitos e o tamanho da equipe. A XP é um ciclo de vida de desenvolvimento voltado para projetos com equipes pequenas, sistemas orientados a objeto e com desenvolvimento incremental, segundo Teles (2014), sendo dividido em iterações que evoluem a aplicação de forma incremental. Primeiramente é definido o escopo e as ferramentas a serem utilizadas. Então há o desenvolvimento do protótipo e, por fim, o desenvolvimento da solução planejada.

2.1 Ferramentas Utilizadas

Antes do início da implementação da solução, foram escolhidas as ferramentas a serem utilizadas no projeto, de acordo com os requisitos levantados. Durante o desenvolvimento, essas serão as ferramentas utilizadas:

1. GitHub[1]: plataforma para hospedagem de código-fonte que utiliza o Git como sistema de controle de versão. Será utilizado para o armazenamento do código da solução pelo seu caráter aberto e gratuito;

2. C++[2], linguagem de programação compilada multiparadigma. Será utilizada como linguagem de implementação principal pelo suporte à programação orientada a objetos e domínio que a equipe tem dela;

3. Lilypond[3], formato de escrita musical baseado em TeX. Será utilizado para representação da melodia por ter um formato padronizado e menos verboso se comparado a outras opções como o MusicXML. Além disso, seu pacote para Linux conta com programas como midi2ly e musicxml2ly, que permite converter músicas de formatos populares para o formato .ly padrão do Lilypond;

4. MuseScore[4], editor de partituras. Será utilizado para criação e edição de partitura pelo fato da equipe ter domínio sobre a ferramenta e ser compatível com Linux e Windows, ao contrário de alternativas similares como o Encore.

[1] <https://github.com/>
[2] <http://www.cplusplus.com/>
[3] <http://lilypond.org/>
[4] <https://musescore.com/>

2.2 Ciclo de Vida de Desenvolvimento

O ciclo de vida de desenvolvimento contará com três fases bem definidas. Primeiramente, serão levantados os requisitos. Então, serão escolhidas as ferramentas a serem utilizadas. Após isso, será desenvolvido um protótipo focado em contrapontos de primeira espécie. Os resultados do protótipo guiarão o desenvolvimento posterior, que será focado na composição algorítmica de contrapontos palestrinianos de até quinta espécie.

2.2.1 Requisitos

Os requisitos foram desenvolvidos na primeira parte do trabalho. O escopo inicial era de um *software* capaz de ler uma melodia monofônica e gerar contrapontos palestrinianos de até quinta espécie para ela no formato Lilypond. Veja um fluxograma simplificado da aplicação na Figura 31.

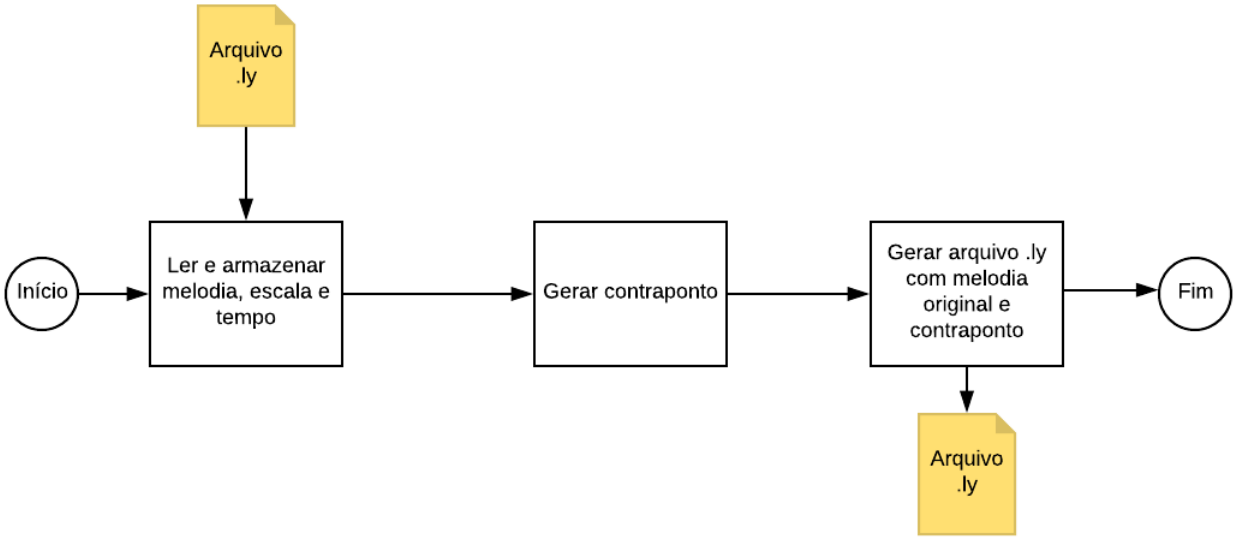

Figura 31 – Fluxograma simplificado da aplicação

Após o estudo da teoria musical necessária para a construção de contrapontos,foram definidos os seguintes módulos a serem desenvolvidos:

1. módulo de notas musicais;
2. módulo de intervalos;
3. módulo de escalas;
4. módulo de contrapontos de primeira espécie;
5. módulo de contrapontos de segunda espécie;
6. módulo de contrapontos de terceira espécie;

7. módulo de contrapontos de quarta espécie;

8. módulo de contrapontos de quinta espécie;

9. módulo de construção do MIDI.

2.2.1.1 Módulo de Notas Musicais

Este módulo é responsável pela leitura e armazenamento das notas de uma melodia. Ele conta com as seguintes capacidades: armazenamento do atributos de uma nota, *parser* de uma nota em formato Lilypond e armazenamento de uma melodia completa. O armazenamento de uma nota inclui estes atributos: a figura musical que representa a duração, o tempo absoluto da nota, os acidentes aplicados a ela, a oitava em que ela está inserida, seu número MIDI e seu número em relação a notas sem contar acidentes. Além disso, essa parte deve ser capaz de devolver notas enarmônicas a uma nota – notas com nomenclatura diferente, mas som e número de semitons (em relação a uma nota qualquer) iguais, como C♯ e D♭.

O *parser* de uma nota deve ser feito a partir de um arquivo Lilypond e armazenado na estrutura responsável pelo armazenamento de notas já citada. Além disso, ele deve ser capaz de retornar uma nota armazenada no mesmo formato lido.

O armazenamento da melodia completa deve ser capaz de armazenar as notas de uma melodia preservando sua ordem, além de armazenar o tempo dos compassos e a escala da música.

2.2.1.2 Módulo de Intervalos

O módulo de intervalos deve ser capaz de armazenar intervalos, gerá-los a partir de duas notas ou de uma *string* e de retornar a outra nota dados uma nota e um intervalo, por exemplo, retornar G♯5 ao receber C5 e um intervalo de quinta aumentada.

O armazenamento de intervalos deve possuir os seguintes atributos: classificações quantitativa e qualitativa, número de semitons e se o intervalo é ascendente ou não (considerando a nota recebida como a primeira).

O método de retorno de uma nota, dado uma nota e um intervalo deve ser capaz de retornar a segunda nota para qualquer intervalo definido, de forma que a distância entre as duas notas seja compatível com a classificação quantitativa e qualitativa do intervalo. Ele também deve ser capaz de indicar se é impossível gerar tal nota e retornar uma nota em uma dada escala previamente definida, quando necessário.

2.2.1.3 Módulo de Escalas

O módulo de escalas deve ser capaz de armazenar uma escala e responder se uma nota faz ou não parte dela. O armazenamento da escala deve ser capaz de armazenar quais notas estão presentes nela a partir da primeira nota e do modo (maior ou menor) ou da primeira nota e de um conjunto de intervalos.

O objeto de uma escala deve ser capaz de receber um objeto de nota e dizer se aquela nota está ou não presente, baseada em seu número de MIDI e sua representação, por exemplo, a nota B está presente na escala de Dó Maior, mas não a nota C♭, embora elas sejam enarmônicas.

2.2.1.4 Módulo de Contraponto

Cada módulo deve ser capaz de gerar um contraponto palestriniano de acordo com as regras definidas pela espécie do contraponto. Eles devem receber uma melodia monofônica e gerar um contraponto daquela espécie de acordo com os parâmetros fornecidos em relação a número de movimentos reversos, consonâncias condicionais paralelas e outros parâmetros relacionados a regras que não sejam exatas. Como exemplo, tem-se a regra que se evita, mas não se proíbe movimentos paralelos, cabendo ao usuário definir o número de movimentos paralelos aceitáveis. Com tais parâmetros, o módulo deve ser capaz de gerar um contraponto aleatório para cada iteração do algoritmo.

2.2.1.5 Módulo de Construção do MIDI

O módulo de construção do MIDI é um módulo que deve automatizar a construção de um MIDI tendo como base o arquivo Lilypond original e o contraponto gerado. Com esses dois, esse módulo deve inserir o contraponto em uma cópia do arquivo original e gerar o MIDI utilizando o pacote do Lilypond para Linux.

2.2.2 Protótipo

Assim como os requisitos, o protótipo também foi desenvolvido na primeira parte do trabalho. O protótipo possui a finalidade de testar o conceito da solução por meio um MVP (produto mínimo viável ou *minimum viable product*). O protótipo proposto possuía as seguintes funcionalidades:

1. módulo de notas musicais, incluindo leitura de arquivo Lilypond simplificado e armazenamentos de notas de uma melodia;

2. módulo de intervalos, incluindo a construção de intervalos a partir de notas ou *strings* padronizadas e retorno de nota ao se receber nota e intervalo;

3. módulo de escalas, incluindo construção de escalas a partir do modo ou de um conjunto de intervalos;

4. módulo de contrapontos de primeira espécie implementado utilizando busca completa;

5. módulo de contrapontos de primeira espécie implementado utilizando programação dinâmica;

6. módulo de contrapontos de segunda espécie, com leitura de tempo de compasso.

O protótipo contém os módulos de notação, intervalos e escalas pois são considerados módulos básicos para a construção de contrapontos.

O módulo de contrapontos de primeira espécie com algoritmo de busca completa pretende testar se a geração de contrapontos de forma *naive* (a implementação mais simples possível) é correta e computacionalmente viável.

O módulo de contrapontos de segunda espécie com algoritmo de DP pretende testar se a programação dinâmica efetivamente diminui o tempo de execução do algoritmo e se essa diminuição é relevante o suficiente para justificar o uso desse paradigma.

O módulo de contrapontos de segunda espécie busca validar a corretude do módulo de primeira espécie uma vez que tenha sido escalado. Dentre as novas dificuldades, há o fato de que há duas notas no contraponto para cada do *cantus firmus*, a posição da nota no compasso possui papel decisivo no algoritmo, invalidando a estratégia de avaliar os intervalos nota a nota e adicionando um estado à DP.

2.2.3 Desenvolvimento

Após a definição dos requisitos e a construção do protótipo, o sistema terá o restante de suas funcionalidades implementadas e testadas. As atividades previstas para a fase de desenvolvimento são:

1. testes dos módulos de notação, intervalos e escalas;

2. testes dos módulos de contraponto de primeira e segunda espécie;

3. otimização do *parser*;

4. implementação e testes do módulo de contrapontos de terceira espécie;

5. implementação e testes do módulo de contrapontos de quarta espécie;

6. implementação e testes do módulo de contrapontos de quinta espécie;

7. implementação e testes do módulo de construção de MIDI.

A segunda parte do trabalho, que engloba o desenvolvimento, começará com testes das funcionalidades implementadas no protótipo. O objetivo é garantir que a solução do protótipo é adequada e completa antes de começar a implementação definitiva, refatorando o código e removendo *bugs* identificados.

Após isso, há a otimização do *parser*. O *parser* atual lê apenas um arquivo Lilypond modificado para conter apenas o tempo, a escala e as notas. Pretende-se criar um *parser* que, ainda que simples, seja capaz de ler arquivos Lilypond sem tais modificações.

Seguindo a otimização do *parser*, haverá a implementação e teste dos contrapontos de terceira, quarta e quinta espécie. Baseado no código já implementado dos contrapontos de primeira e segunda espécie, eles serão desenvolvidos sequencialmente, já que as novas regras de uma espécie somam-se às regras da espécie anterior.

Por fim, haverá a implementação e teste de um módulo de construção de MIDI com as especificações definidas na subseção 2.2.1.

2.3 Atividades e Cronograma

O desenvolvimento do sistema terá as seguintes atividades:

1. definição dos requisitos do sistema;
2. estudo sobre teoria musical, incluindo notação musical, intervalos, escalas e contrapontos;
3. implementação do protótipo de parser de um arquivo Lilypond;
4. implementação do módulo de notas musicais;
5. implementação do módulo de intervalos;
6. implementação do módulo de escalas;
7. implementação do protótipo do módulo de contrapontos;
8. implementação do módulo de contrapontos de primeira espécie utilizando busca completa;
9. implementação do módulo de contrapontos de primeira espécie utilizando programação dinâmica;
10. escrita do TCC 1;

11. implementação do módulo de contrapontos de segunda espécie;
12. testes do *parser*;
13. testes do módulo de notas musicais;
14. testes do módulo de intervalos;
15. testes do módulo de escalas;
16. testes dos módulos de contrapontos de primeira e segunda espécie;
17. otimização do *parser*;
18. implementação e testes do módulo de contrapontos de terceira espécie;
19. implementação e testes do módulo de contrapontos de quarta espécie;
20. implementação e testes do módulo de contrapontos de quinta espécie;
21. implementação e testes do módulo de construção de MIDI;
22. escrita do TCC 2.

O cronograma do trabalho está apresentado na Figura 32.

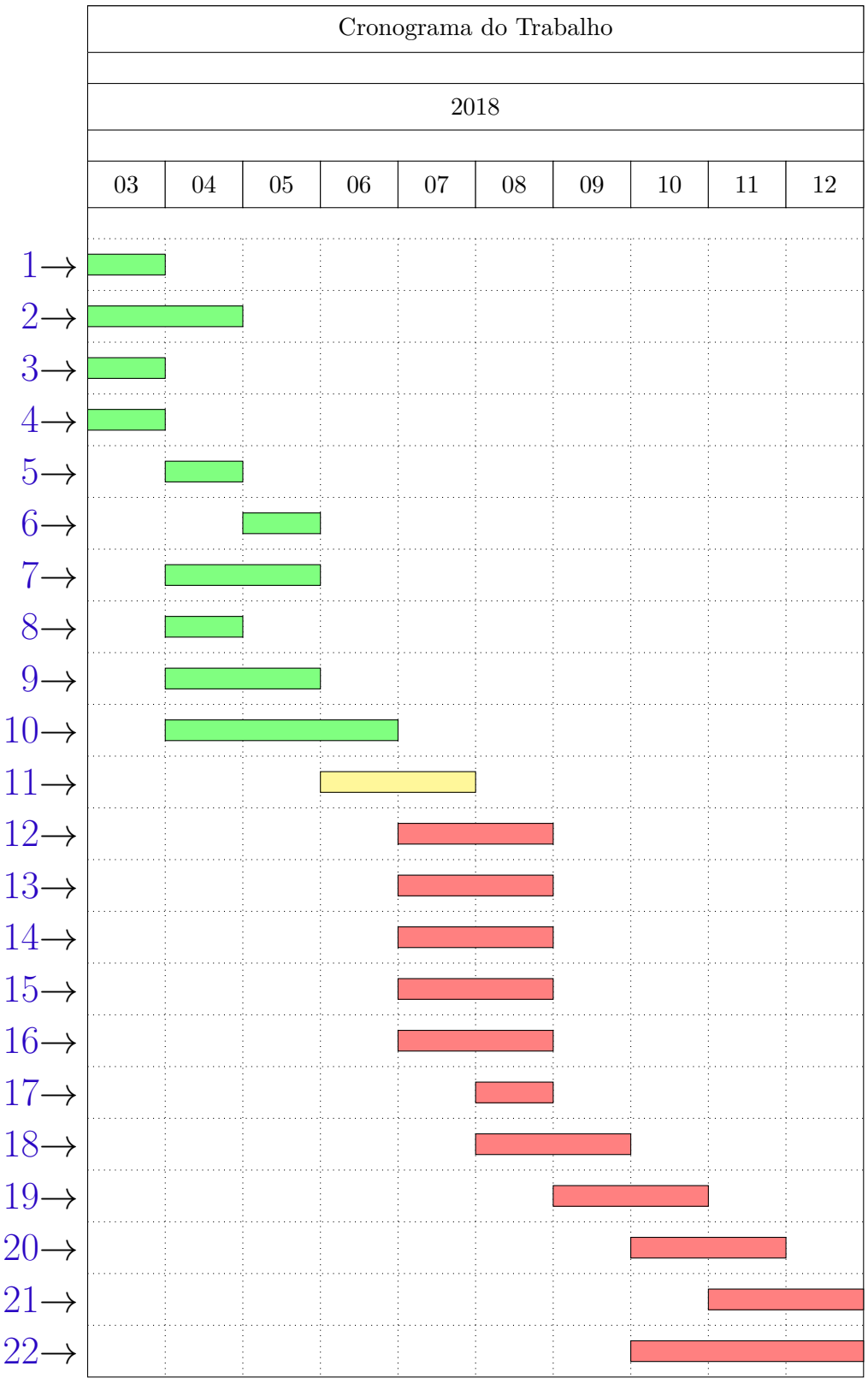

Figura 32 – Cronograma

3 Resultados Obtidos

Na primeira parte desse trabalho, foi implementado um protótipo com as condições mínimas para a execução do algoritmo de composição de contrapontos de primeira espécie. Para isso, foram implementados os módulos de notação musical, intervalos, escalas e contraponto. A execução dos testes é feita em uma função *Main*.

3.1 Módulo de Notação Musical

O módulo de notação musical possui quatro classes: *Note*, *Note Reader*, *Compass Time* e *Song*.

3.1.1 *Note*

A classe *Note* é responsável pelo armazenamento das informações de uma nota. Os atributos de instância dessa classe são:

1. *Duration*: número inteiro que representa a quantidade de tempo que a nota é tocada dentro de um compasso.
2. *Absolute Time*: número inteiro que indica a posição absoluta da nota na melodia, importante para contrapontos de segunda espécie em diante que utilizam da posição da nota no compasso.
3. *Note*: *string* que armazena qual nota musical, entre A e G, ou R quando for uma pausa.
4. *Accidental*: *string* que armazena os acidentes musicais que ocorrem à nota, se houver.
5. *Octave*: número inteiro que indica qual de qual oitava a nota é.
6. *MIDI Number*: número inteiro que representa a nota no formato padronizado MIDI, por exemplo, o Dó na quarta oitava é representado pelo número 72.
7. *Note Number*: número inteiro que representa a nota desconsiderando acidentes, utilizado para a classificação quantitativa de intervalos.
8. *Valid*: atributo booleano que indica se a nota é válida ou não, útil para retornar notas inválidas sem utilizar ponteiros, quando necessário.

Além de construtores, a classe possui os seguintes métodos:

1. *Full Note*: retorna uma *string* representando a nota.

2. *Set Full Note*: constrói os outros atributos da uma nota a partir de uma *string* padronizada, utilizada em construtores e outros métodos.

3. *Enarmonies*: retorna um vetor com as notas enarmônicas à nota do objeto.

Por ser um protótipo, todos os atributos e métodos são públicos, isso será mudado na versão final, a Figura 33 descreve os atributos, métodos e construtores da classe.

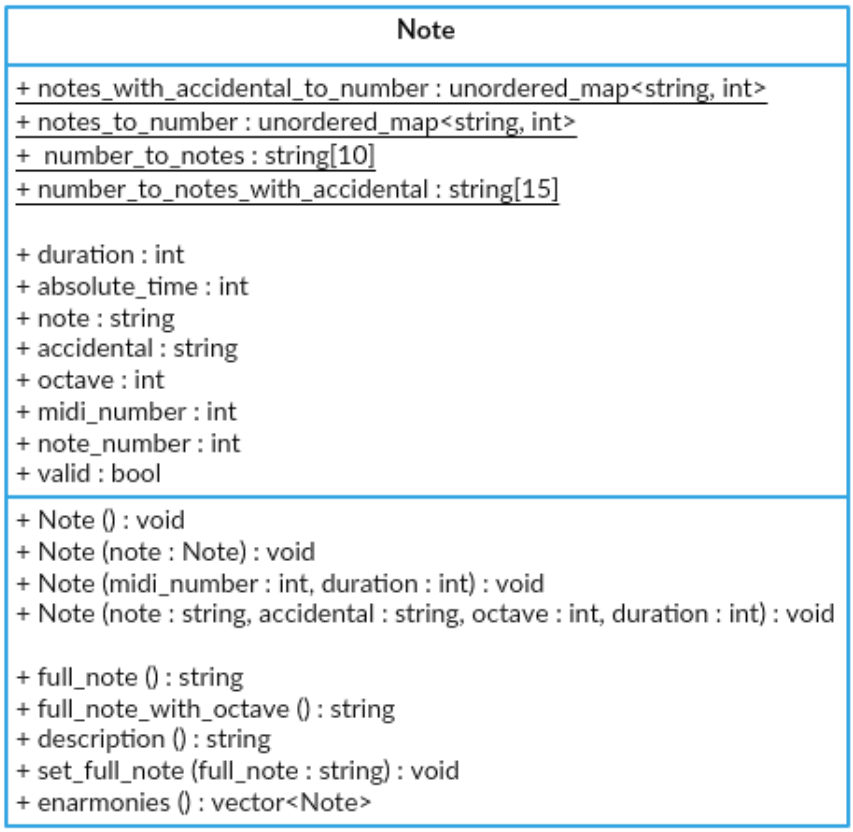

Figura 33 – Diagrama da Classe *Note*

3.1.2 *Note Reader*

A classe *Note Reader* é responsável pelo *parser* de *string* representado notas em formato Lilypond para instâncias da classe *Note* e vice-versa. Sem a necessidade de atributos e com apenas métodos de classe, a classe age como uma biblioteca, sendo utilizada sem instanciação. Ela possui os seguintes métodos:

1. *String To Note*: retorna uma instância de *Note* ao receber uma *string* e uma nota. A nota recebida como parâmetro é a nota anterior, pois o formato Lilypond omite a duração da nota se for a mesma da anterior.

2. *Note To String*: retorna uma *string* que representa uma nota em formato Lilypond ao receber um objeto *Note*.

3. *MSB*: retorna o bit mais significativo, utilizado no cálculo da figura musical baseado na duração da nota. Quanto maior for o valor da figura musical, menor a duração, mas pontos de aumento podem modificar a duração da nota e o bit mais significativo é usado para inferir o valor da figura musical, pois as figuras sem pontos de aumento sempre geram durações representadas por potências de 2.

4. *Number Of On Bits*: retorna o número de bits ligados de um inteiro, útil pois deriva o número de pontos de aumento de uma nota (para gerar a *string*) a partir da representação binária da duração.

A Figura 34 representa os métodos da classe.

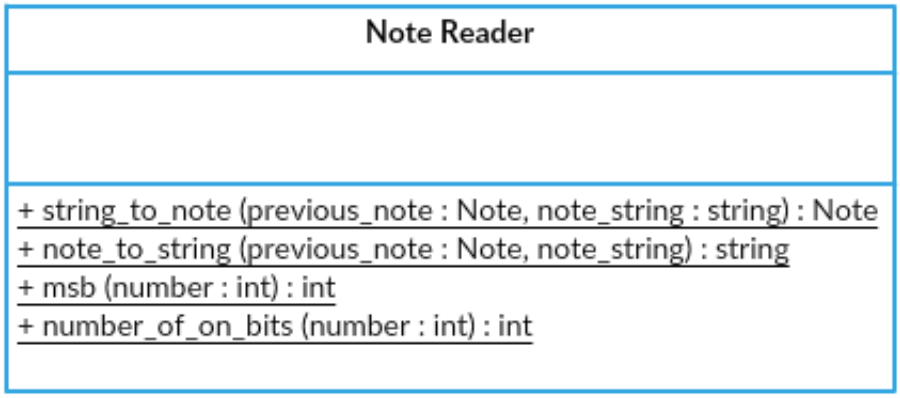

Figura 34 – Diagrama da Classe *Note Reader*

3.1.3 *Compass Time*

A classe *Compass Time* armazena o tempo de compasso de uma música. Possui os seguintes atributos:

1. *Times*: Número inteiro, armazena a quantidade de batidas que o compasso possui, por exemplo, em um compasso 3/4, ele armazena o valor 3.

2. *Base Note*: Número inteiro, armazena qual é a figura musical base do compasso, por exemplo, em um compasso 3/4, ele armazena o valor 4.

Além dos atributos, essa classe possui dois métodos:

1. *Base Note Duration*: Retorna a duração da nota base.

2. *Compass Duration*: Retorna a duração total do compasso, multiplicando a duração da nota base pela quantidade de batidas.

A Figura 35 representa os atributos e métodos da classe.

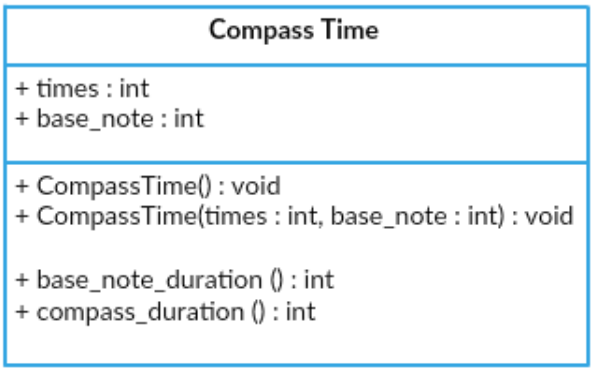

Figura 35 – Diagrama da Classe *Compass Time*

3.1.4 *Song*

A classe *Song* armazena as informações básicas de uma música. Possui os seguintes atributos:

1. *Notes*: Vetor que armazena as notas de uma música em ordem de execução.
2. *Scale*: Armazena a escala de uma música.
3. *Compass Time*: Armazena o tempo de compasso de uma música, assumindo que não há troca de tempo no meio da canção.

Ela possui apenas um método chamado *Size* que retorna o número de notas da música. A Figura 36 representa os atributos e métodos da classe.

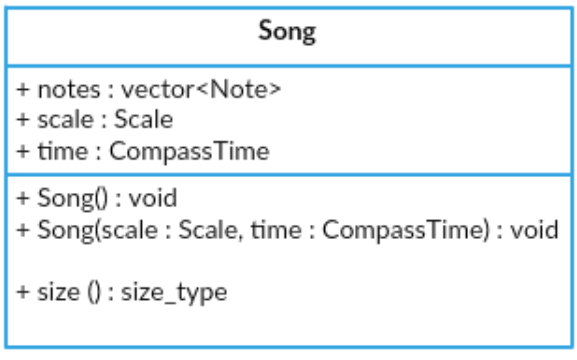

Figura 36 – Diagrama da Classe *Song*

3.2 Módulo de Intervalos

Responsável pela geração e armazenamento de intervalos, este módulo possui a classe *Interval*.

3.2.1 *Interval*

A classe *Interval* armazena intervalos e possui método para gerar uma nota a partir de outra e um intervalos. Ela possui os seguintes atributos:

1. *Quantitative*: número inteiro, representa a classificação quantitativa do intervalo.
2. *Qualitative*: *string* que representa a classificação qualitativa do intervalo.
3. *Half Tones*: número inteiro, representa a quantidade de semitons característica do intervalo.
4. *Ascendant*: booleano que indica se o intervalo é ou não ascendente, no caso de intervalos melódicos.

Excetuando-se métodos de impressão de atributos, a classe possui os seguintes métodos:

1. *Is Dissonant*: retorna um booleano definindo se o intervalo é ou não dissonante, baseado nas regras do Contraponto Palestriniano.
2. *Is Consonant*: retorna um booleano definindo se o intervalo é ou não consonante.
3. *Interval to Note*: método de classe que retorna uma nota ao receber uma nota e um intervalo. A nota recebida é considerada a primeira ao utilizar a propriedade do intervalo ser ascendente ou descendente, pois uma nota e um intervalo sem atributo de ascendência pode gerar até duas notas, uma acima e outra abaixo da nota original.

A Figura 37 representa os atributos e métodos da classe.

3.3 Módulo de Escalas

O módulo de escalas é responsável pela geração e armazenamento de escalas musicais. Ele possui uma classe, a *Scale*.

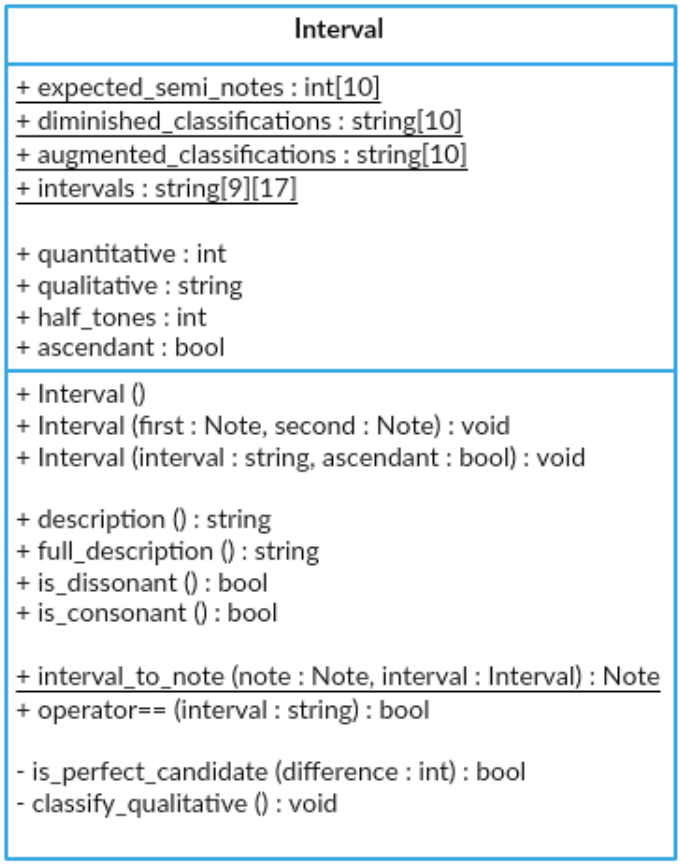

Figura 37 – Diagrama da Classe *Interval*

3.3.1 *Scale*

A classe *Scale* armazena as notas de uma escala, possuindo apenas um atributo, um *set* de *strings* chamado *Permitted Notes* que define as notas permitidas para aquela escala. Além disso, ela possui os seguintes métodos:

1. *Is Valid Note*: recebe uma nota e diz se ela está ou não presente na escala.

2. *Interval To Note On Scale*: método de classe que retorna uma nota, dado uma nota, um intervalo e uma escala. Retorna inválido se a nota não estiver presente na escala.

A Figura 38 representa os atributos e métodos da classe.

3.4 Módulo de Contraponto

O módulo de contraponto é responsável pela geração de contrapontos. Utilizando todas as classes anteriormente citadas, atualmente possui duas classes: *Counterpoint* e *First Order Counterpoint*.

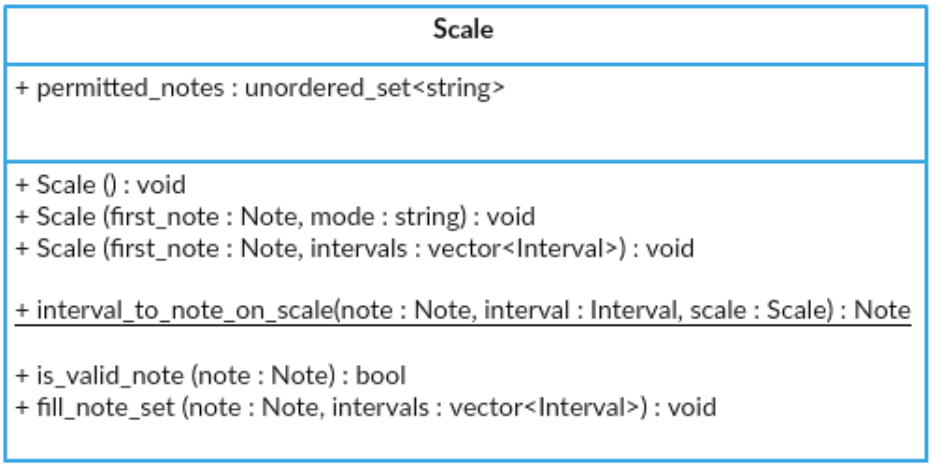

Figura 38 – Diagrama da Classe *Scale*

3.4.1 *Counterpoint*

A classe *Counterpoint* funciona como uma biblioteca e não possui atributos de instância. Inicialmente utilizada para a implementação do contraponto de primeira ordem, atualmente possui o método ad-hoc de geração de contrapontos e o método de análise de notas para inserção no contraponto chamado *Analyse and Add Interval*.

A solução inicialmente concebida utiliza um algoritmo de seleção aleatório. Iniciando na primeira nota do *cantus firmus*, monta-se um vetor de notas possíveis para o contraponto em seu estado atual, escolhe-se aleatoriamente uma das notas do vetor e a insere no vetor do contraponto, seguindo para a próxima nota. O vetor de notas possíveis era montado de acordo com as regras do contraponto de primeira espécie anteriormente citadas.

O problema dessa solução era o fato de escolher randomicamente um caminho e não explorar os outros, o que podia levar a um caminho que não levava a uma solução. Para contornar isso, o algoritmo era rodado diversas vezes até se encontrar uma solução. Contudo, em músicas maiores o tempo de execução chegava a dois minutos ou mais, e em músicas impossíveis de se gerar contraponto, o algoritmo entrava em *loop* infinito.

Atualmente, essa classe é utilizada como classe pai da *First Order Counterpoint* e será pai das classes dos contrapontos de outras espécies. A Figura 39 apresenta os métodos da classe.

3.4.2 *First Order Counterpoint*

A classe *First Order Counterpoint* herda da classe *Counterpoint* e é uma biblioteca, possuindo atributos e métodos de classe. Possui os seguintes métodos:

1. *DFS Generate Counterpoint*: recebe uma instância de *Song*, um booleano que define

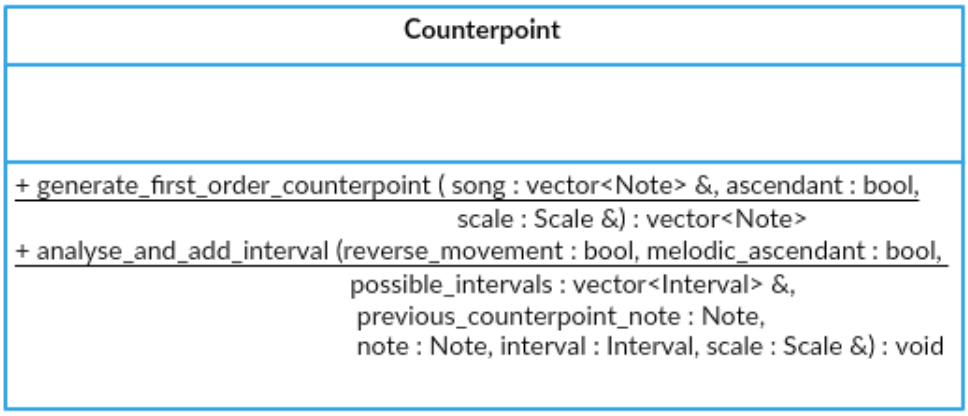

Figura 39 – Diagrama da Classe *Counterpoint*

se o contraponto será superior ou inferior e os números de terças e sextas paralelas e movimentos paralelos permitidos. Retorna um vetor de notas correspondente ao contraponto gerado.

2. *Solve*: Método recursivo que executa a busca completa para gerar contrapontos. Tem como parâmetros a posição atual no *cantus firmus*, o número atual de terças ou sextas paralelas, o número de movimentos paralelos já ocorridos, a instância do *cantus firmus* e da proposta de contraponto atual passadas por referência e um booleano indicando se o contraponto é inferior ou superior.

O método que retorna o vetor de contrapontos encapsula a solução e chama o método recursivo. O algoritmo funciona da seguinta forma, iniciando na primeira posição do *cantus firmus*, o estado da DP é definido por quatro parâmetros: posição, número MIDI da nota atual, número de terças ou sextas paralelas anteriormente executadas e número de movimentos paralelos.

A transição ocorre após a geração do vetor de notas possíveis, de acordo com as regras do contraponto. Para cada nota do vetor de intervalos, adiciona-se a nota ao final do vetor de contrapontos se a inserção dela não fizer ultrapassar o número de terças, sextas e movimentos paralelos. Após a adição da nota, o método *Solve* é chamado para a posição mais um, com o número de terças, sextas e movimentos paralelos atualizados. O retorno do método *Solve* é um booleano que indica se há um contraponto válido ao explorar aquele caminho. Portanto, se o retorno for *false*, a nota adicionada no final do vetor de contrapontos é removida e a próxima nota do vetor de notas possíveis é testada. Contudo, se o retorno for *true*, ela também retorna *true* e não remove a nota atual, mantendo o vetor de contraponto gerado como solução. Se a análise das notas não adicionar nenhuma nota no vetor de notas possíveis, o método retorna também *false*. Vale ressaltar que o vetor de notas possíveis é embaralhado aleatoriamente para garantir uma solução diferente a cada execução do algoritmo. Além disso, primeiro adiciona-se as notas que geram movimento

contrário, embaralha-se a primeira parte do vetor, em seguida, adiciona-se as notas que geram movimento paralelo e embaralha-se a segunda parte do vetor. Isso é feito para que a solução priorize o menor número de movimentos paralelos possível.

O caso-base da DP é quando a posição é maior que o tamanho do *cantus firmus*. Se o algoritmo chegar a essa posição, significa que a proposta de contraponto atual é um contraponto válido, portanto, retorna-se *true*.

Para que o algoritmo funcione efetivamente como uma DP, é utilizado um vetor de memorização, sendo um atributo de classe, é um vetor booleano de quatro dimensões, uma para cada estado. A primeira possui 201 espaços, permitindo que um contraponto seja gerado para um *cantus firmus* com, no máximo, 201 notas. A segunda dimensão possui 90 espaços e representa o número MIDI da nota atual e cobre as 88 primeiras notas do piano. A terceira dimensão possui 5 espaços e cobre as 4 terças ou sextas paralelas seguidas permitidas pelo algoritmo. A quarta dimensão representa o número de movimentos paralelos que aconteceram durante o contraponto, possui 100 espaços pois o máximo de movimentos paralelos permitidos pelo algoritmo é metade do tamanho do contraponto ou menos.

Utilizando o vetor de DP, ele inicia todas as posições com *true*. Quando o algoritmo encontra um estado que é não possui solução, memoriza como *false* na posição do vetor de DP que representa aquele e todas as posteriores consultas àquele estado retornam falso sem explorar esse candidato de solução novamente. Um exemplo é se o algoritmo definir que o estado atual – na posição 10, na nota C4 (*MIDI number* 60), com 2 terças ou sextas paralelas e 13 movimentos paralelos – não é uma solução, memoriza-se *false* em dp[10][60][2][13]. Na próxima consulta a esse estado, retorna falso imediatamente sem explorar. Memorizar estados que retornam *true* é irrelevante pois o primeiro estado completo que chega ao final retornando *true* é definido como a solução.

A Figura 40 apresenta os métodos e atributos da classe.

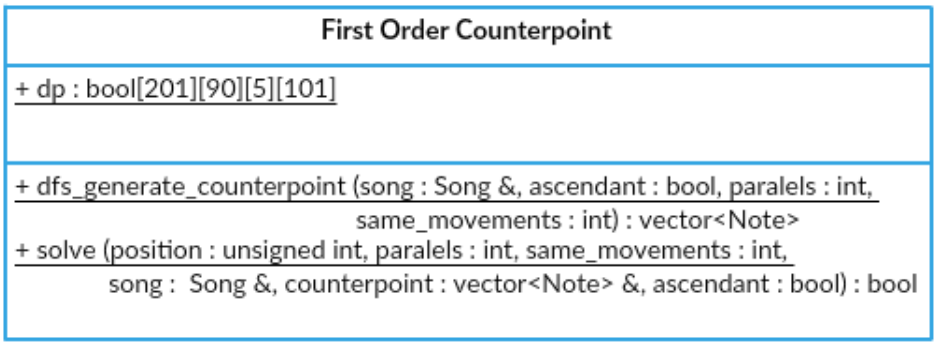

Figura 40 – Diagrama da Classe *First Order Counterpoint*

3.5 *Main*

A *Main* é a função principal que executa o programa. Nela o arquivo .ly é lido – primeiramente lendo a tônica e o modo da escala, seguido pela leitura do ritmo. Após isso, cada nota é lida e convertida para uma instância de *Note* por meio da *Note Reader*. Para efeitos de teste, atualmente, os intervalos melódicos são calculados e impressos no terminal, bem como a transformação de *Note* para *string* novamente.

Após a leitura, o algoritmo de contraponto é executado. É gerado um vetor de notas com o contraponto de primeira espécie. Esse vetor é impresso no terminal em notação musical padrão e em notação específica do Lilypond para fácil transferência do contraponto para o arquivo original.

3.6 Experimentos

Para validar os algoritmo construído, foram feitos experimentos em músicas pequenas para melhor análise do resultado. O teste da execução do algoritmo foi feito utilizando várias músicas. Serão apresentados os testes feitos com a música infantil *Twinkle Twinkle Little Star*. O código-fonte da solução está disponível em *<https://github.com/joao18araujo/ly_parser>*.

A música original pode ser observada em formato Lilypond e em pártitura no código 3.1 e na figura 41, respectivamente.

Código 3.1 – *Twinkle Twinkle Little Star*

```
\key c \major
\time 4/4
c''4 c'' g'' g''
a'' a'' g''2
f''4 f'' e'' e''
d'' d'' c''2
g''4 g'' f'' f''
e'' e'' d''2
g''4 g'' f'' f''
e'' e'' d''2
c''4 c'' g'' g''
a'' a'' g''2
f''4 f'' e'' e''
d'' d'' c''2
```

Após a primeira execução do algoritmo, o contraponto gerado pode ser visto em formato Lilypond no código 3.2 e em partitura na Figura 42.

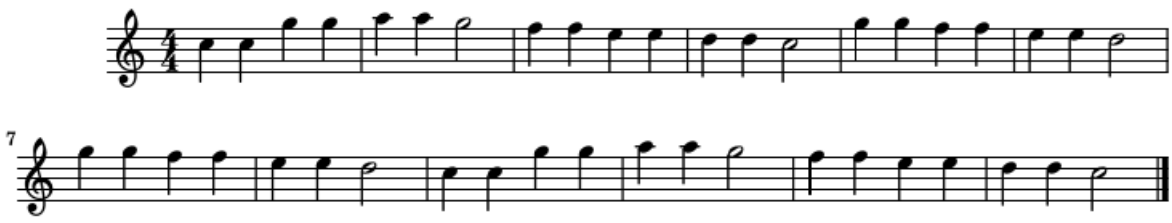

Figura 41 – Partitura de *Twinkle Twinkle Little Star*

Código 3.2 – Primeiro contraponto gerado

```
c''4 c'4 e'4 e''4
c''4 f''4 e''2 d''4
d''4 c''4 g'4 b'4
f'4 a'2 g'4 b'4
d''4 d'4 c''4 g'4
b'2 g'4 b'4 d''4
d'4 c''4 g'4 d'2
a'4 e'4 e'4 b'4
a'4 c''4 g'2 d''4
d'4 e'4 g'4 b'4
f'4 c''2
```

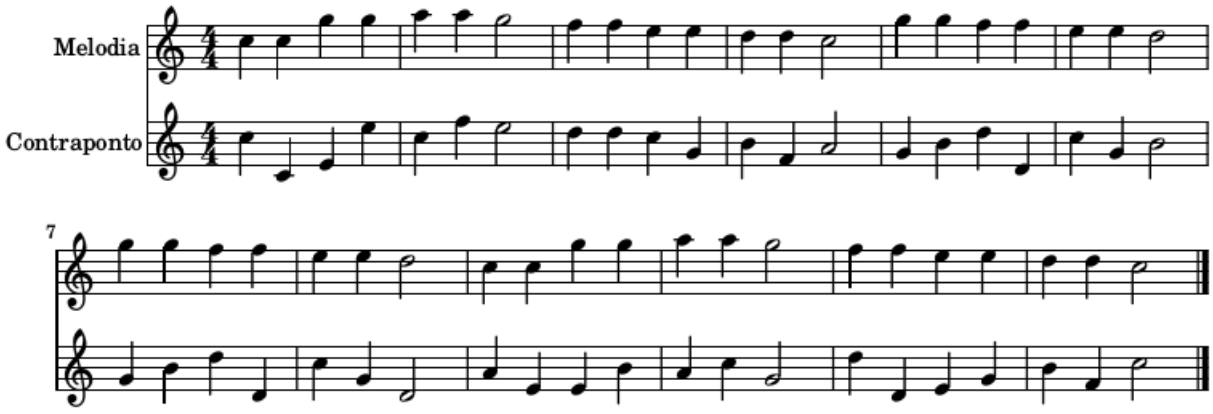

Figura 42 – Partitura do primeiro contraponto gerado

Após a segunda execução do algoritmo, o contraponto gerado pode ser visto em formato Lilypond no código 3.3 e em partitura na Figura 43.

Código 3.3 – Segundo contraponto gerado

```
c'4 e'4 e'4 e''4
f'4 f''4 e''2 d''4
d''4 c''4 g'4 b'4
```

```
f'4 a'2 e'4 g'4
d''4 d'4 g'4 e'4
f'2 e'4 g'4 d''4
d'4 g'4 e'4 f'2
a'4 e'4 e'4 e''4
a'4 c''4 g'2 a'4
f'4 g'4 e'4 b'4
f'4 c''2
```

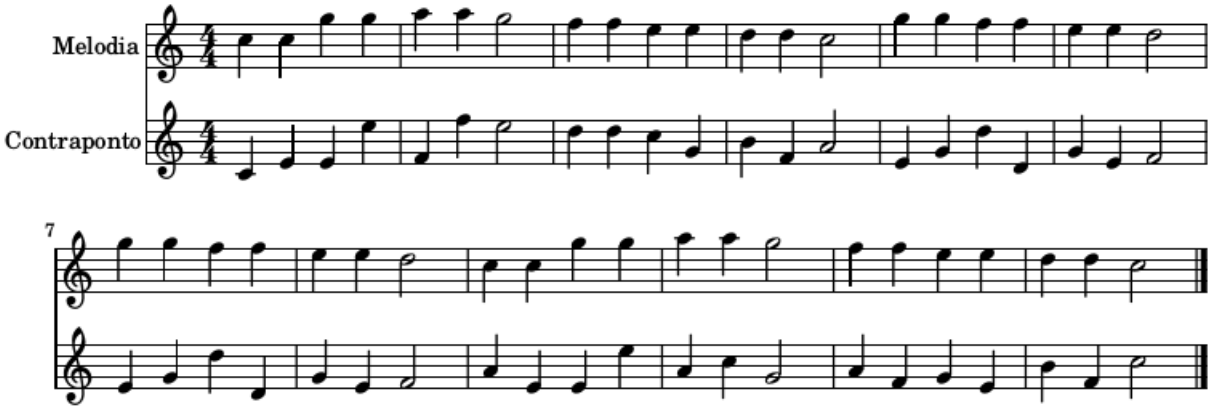

Figura 43 – Partitura do segundo contraponto gerado

Após a terceira execução do algoritmo, o contraponto gerado pode ser visto em formato Lilypond no código 3.4 e em partitura na Figura 44.

Código 3.4 – Terceiro contraponto gerado

```
c''4 a'4 e'4 e''4
f'4 a'4 b'2 d''4
d''4 c''4 g'4 b'4
f'4 a'2 g'4 b'4
d''4 a'4 c''4 g'4
b'2 g'4 b'4 d''4
d'4 c''4 c'4 d'2
e'4 a'4 e'4 b'4
a'4 c''4 e''2 d''4
d'4 c''4 c'4 b'4
b'4 c''2
```

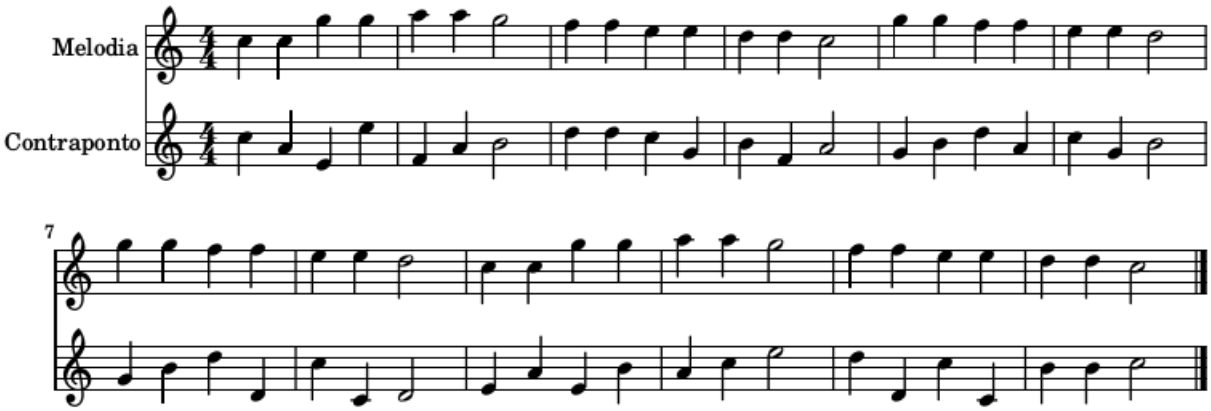

Figura 44 – Partitura do terceiro contraponto gerado

4 Considerações Finais

A partir de regras bem definidas e a representação de música de uma forma digital, foi possível gera contrapontos de primeira espécie palestrinianos algoritmicamente. Embora o *parser* ainda possua caráter básico e o resultado ainda seja impresso no terminal, a primeira parte do trabalho foi concluída com sucesso. Os módulos de notação musical, escala e intervalo estão completos e funcionais. O módulo do contraponto de primeira espécie foi concluído e oferece uma boa base para os contrapontos de outras espécies.

4.1 Trabalhos Futuros

Em relação aos trabalhos futuros, as seguintes tarefas podem ser listadas:

1. Resolver *bugs* da implementação incompleta do contraponto de segunda espécie, ele já é gerado, mas fere algumas regras mais complexas.
2. Testar e refatorar as classes já implementadas para aumentar a solidez do código.
3. Otimizar o *parser* para receber um arquivo Lilypond completo e inserir em uma cópia dele o contraponto gerado.
4. Implementar os módulos de contrapontos de terceira, quarta e quinta espécie.
5. Implementar o módulo de construção do MIDI do *cantus firmus* com o contraponto.

Referências

GHOSE, D. An algorithm for a digital audio format to simulate improvisations in recorded music. *2014 Int. Conf. IMpact E-Technology US, IMPETUS 2014*, p. 29–33, 2014. Citado na página 21.

GOGAL, M.; GOGA, N. Feelings Based Computer Music. p. 673–676, 2004. ISSN 08407789. Citado na página 21.

HALIM, S. *Competitive Programming 3*. Singapura, 2013. 447 p. Citado 3 vezes nas páginas 34, 37 e 38.

JEPPESEN, K. *Counterpoint: the polyphonic vocal style of the Sixteenth Century*. Nova York, Estados Unidos da América, 1992. 320 p. Citado 3 vezes nas páginas 26, 32 e 33.

KOELLREUTER, H.; JOACHIM, H. *Contraponto Modal do Século XVI (Palestrina)*. Brasília, Brasil, 1996. 85 p. Citado na página 32.

MED, B. *Teoria da Música*. Brasília, Brasil, 1996. 420 p. Citado 2 vezes nas páginas 23 e 27.

TELES, V. M. *Extreme Programming*. Rio de Janeiro, Brasil, 2014. 328 p. Citado na página 41.

TRAGTENBERG, L. *Contraponto: Uma Arte de Compor*. São Paulo, Brasil, 1961. 266 p. Citado na página 21.