



Universidade de Brasília – UnB
Faculdade UnB Gama – FGA
Engenharia de *Software*

Pierluigi: 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

**Pierluigi: 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 ob-
tenção 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

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

CDU 02:141:005.6

João Vitor Araujo Moura

Pierluigi: 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, 27 de novembro de 2018

Prof. Dr. Edson Alves da Costa Júnior
Orientador

**Prof. Felipe Duerno do Couto
Almeida**
Convidado 1

Prof. Matheus de Sousa Faria
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 objetivo desse trabalho é desenvolver um *software* capaz de compor algorítmicamente 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. A parte final do desenvolvimento consistiu no teste dos módulos implementados na fase de prototipação, implementação dos módulos de contrapontos de segunda a quarta espécie e construção de um módulo de construção do MIDI do *cantus firmus* com o contraponto. A solução utilizada trata a geração de contrapontos como um grafo implícito, utilizando DFS em conjunto com programação dinâmica. A aplicação desenvolvida é capaz de ler um arquivo Lilypond completo, armazenar sua melodia, armazenar e utilizar intervalos e escalas, gerando contrapontos de primeira a quarta espécie e enviando ao usuário um arquivo compactado contendo o arquivo Lilypond, o arquivo MIDI e o PDF com a partitura.

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. 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 species counterpoints. The final stage of development consisted of testing previously implemented modules, implementation of modules to generate second to fourth species counterpoint and development of a MIDI generation module that returns the MIDI of the original song with the counterpoint. The application has a solution that uses an implicit graph along with DFS and dynamic programming. The developed software is capable of reading a full Lilypond file, store its melody, store and use intervals and scales, generating counterpoints of first to second species and sending to the user a compacted file containing the Lilypond file, the MIDI file and the PDF with the sheet.

Key-words: algorithmic composition. complete search. graphs. DFS. DP. dynamic programming. palestrina style counterpoints.

Lista de ilustrações

Figura 1 – Pauta musical	21
Figura 2 – Notas musicais na clave de Sol	22
Figura 3 – Exemplos de compasso e seus respectivos tempos fortes e fracos	23
Figura 4 – Acidentes Musicais	23
Figura 5 – Armaduras de Clave	23
Figura 6 – Exemplo de Primeira Justa	24
Figura 7 – Exemplo de Quarta Justa	25
Figura 8 – Exemplo de Quinta Justa	25
Figura 9 – Exemplo de Oitava Justa	25
Figura 10 – Exemplos de Segunda Menor e Segunda Maior	25
Figura 11 – Exemplos de Terça Menor e Terça Maior	26
Figura 12 – Exemplos de Sexta Menor e Sexta Maior	26
Figura 13 – Exemplos de Sétima Menor e Sétima Maior	26
Figura 14 – Exemplos de Intervalos Aumentados	27
Figura 15 – Exemplos de Intervalos Diminutos	27
Figura 16 – Exemplo de Intervalo Composto	28
Figura 17 – Escala Cromática	28
Figura 18 – Exemplo de Escala Diatônica	29
Figura 19 – Escala de Dó Maior	29
Figura 20 – Escala de Fá Maior	29
Figura 21 – Escala de Lá Menor	29
Figura 22 – Escala de Mi Menor	30
Figura 23 – Exemplo de Contraponto de Primeira Espécie	31
Figura 24 – Exemplo de Contraponto de Segunda Espécie	32
Figura 25 – Exemplo de Contraponto de Terceira Espécie	32
Figura 26 – Exemplo de Cambiata	32
Figura 27 – Exemplo de Contraponto de Quarta Espécie	33
Figura 28 – Exemplo de Notas Possíveis Representados em Grafo	34
Figura 29 – Exemplos de Grafo Não-Direcionado e Direcionado	34
Figura 30 – Exemplos de Grafo Não-DAG, DAG e Árvore	35
Figura 31 – Exemplo de DFS	35
Figura 32 – Exemplo de BFS	36
Figura 33 – Comparativo entre a abordagem não-DP e DP para o cálculo do $F(5)$	38
Figura 34 – Fluxograma simplificado da aplicação	40
Figura 35 – Diagrama da Classe <i>Note</i>	48
Figura 36 – Diagrama da Classe <i>Compass Time</i>	49

Figura 37 – Diagrama da Classe <i>Song</i>	50
Figura 38 – Diagrama da Classe <i>Interval</i>	52
Figura 39 – Diagrama da Classe <i>Scale</i>	53
Figura 40 – Diagrama da Classe <i>Counterpoint</i>	54
Figura 41 – Diagrama da Classe <i>First Species Counterpoint</i>	54
Figura 42 – Diagrama da Classe <i>Second Species Counterpoint</i>	55
Figura 43 – Diagrama da Classe <i>Third Species Counterpoint</i>	55
Figura 44 – Diagrama da Classe <i>Fourth Species Counterpoint</i>	56
Figura 45 – Algoritmo de Geração de Contrapontos	58
Figura 46 – Diagrama da Classe <i>Song Reader</i>	60
Figura 47 – Diagrama da Classe <i>Ly Parser</i>	61
Figura 48 – Página <i>web</i> do <i>Pierluigi</i>	61
Figura 49 – Partitura de <i>Twinkle Twinkle Little Star</i>	62
Figura 50 – Partitura do contraponto de primeira espécie gerado	63
Figura 51 – Partitura do contraponto de segunda espécie gerado	63
Figura 52 – Partitura do contraponto de terceira espécie gerado	64
Figura 53 – Partitura do contraponto de quarta espécie gerado	64

Lista de tabelas

Tabela 1 – Figuras Musicais	22
---------------------------------------	----

Lista de abreviaturas e siglas

DAG	<i>Direct Acyclic Graph</i>
BFS	<i>Breadth-First Search</i>
DFS	<i>Depth-First Search</i>
DP	<i>Dynamic Programming</i>
MIDI	<i>Musical Instrument Digital Interface</i>
XP	<i>eXtreme Programming</i>

Lista de símbolos

#	Acidente musical sustenido
b	Acidente musical bemol
♮	Acidente musical bequadro

Sumário

	Introdução	19
1	FUNDAMENTAÇÃO TEÓRICA	21
1.1	Teoria Musical	21
1.1.1	Notação Musical	21
1.1.2	Intervalos	24
1.1.2.1	Intervalos Justos	24
1.1.2.2	Intervalos Maiores e Menores	25
1.1.2.3	Intervalos Aumentados e Diminutos	26
1.1.2.4	Intervalos Compostos	27
1.1.3	Escalas	28
1.1.3.1	Escalas Maiores	29
1.1.3.2	Escalas Menores	29
1.1.4	Contraponto	30
1.1.4.1	Contraponto de Primeira Espécie	30
1.1.4.2	Contraponto de Segunda Espécie	31
1.1.4.3	Contraponto de Terceira Espécie	31
1.1.4.4	Contraponto de Quarta Espécie	32
1.2	Grafos	33
1.2.1	Grafos Direcionados Acíclicos	33
1.2.2	Travessia de Grafos	35
1.3	Paradigmas de Solução de Problema	36
1.3.1	Busca Completa	36
1.3.2	Programação Dinâmica	37
2	METODOLOGIA	39
2.1	Ferramentas Utilizadas	39
2.2	Ciclo de Vida de Desenvolvimento	40
2.2.1	Requisitos	40
2.2.1.1	Módulo de Notas Musicais	41
2.2.1.2	Módulo de Intervalos	41
2.2.1.3	Módulo de Escalas	42
2.2.1.4	Módulos de Contraponto	42
2.2.1.5	Módulo de Construção do MIDI	43
2.2.2	Protótipo	43
2.2.3	Desenvolvimento	44

2.3	Atividades	45
3	RESULTADOS	47
3.1	Módulo de Notação Musical	47
3.1.1	<i>Note</i>	47
3.1.2	<i>Compass Time</i>	48
3.1.3	<i>Song</i>	49
3.2	Módulo de Intervalos	50
3.2.1	<i>Interval</i>	50
3.3	Módulo de Escalas	51
3.3.1	<i>Scale</i>	51
3.4	Módulo de Contraponto	52
3.4.1	<i>Counterpoint</i>	52
3.4.2	<i>First Species Counterpoint</i>	53
3.4.3	<i>Second Species Counterpoint</i>	55
3.4.4	<i>Third Species Counterpoint</i>	55
3.4.5	<i>Fourth Species Counterpoint</i>	56
3.5	Main	57
3.6	Módulo de Construção do MIDI	58
3.6.1	<i>Song Reader</i>	58
3.6.2	<i>Ly Parser</i>	60
3.6.3	<i>Counterpoint Generator</i>	61
3.7	Experimentos	61
4	CONSIDERAÇÕES FINAIS	65
4.1	Trabalhos Futuros	66
	REFERÊNCIAS	67

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 algorítmicamente?*

Objetivos

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.

Os objetivos específicos deste trabalho são:

- implementar um algoritmo de leitura e interpretação de formatos digitais de representação de música;
- analisar os *softwares* de composição algorítmica de contrapontos já existentes;

¹ <<http://lilypond.org/>>

² <<https://www.musicxml.com/>>

- implementar um algoritmo de composição de contrapontos que respeite as restrições dos modos litúrgicos definidos.

Estrutura do Trabalho

Este trabalho 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 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 problemas: 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. Na notação musical moderna, 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	$\frac{1}{2}$ ♩ ou 2 ♪
Semínima	♫	4	$\frac{1}{2}$ ♪ ou 2 ♫
Colcheia	♬	8	$\frac{1}{2}$ ♫ ou 2 ♬
Semicolcheia	♭	16	$\frac{1}{2}$ ♬

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.

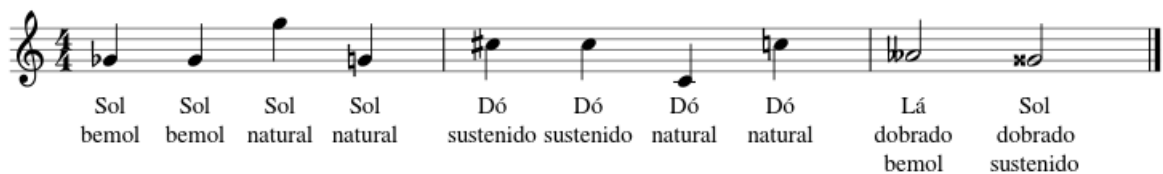


Figura 4 – Acidentes Musicais

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 (6^a), que é a mesma que o intervalo entre G \sharp 4 (Sol sustenido na quarta oitava) e E \flat 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 (1^aJ ou P1¹) 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).

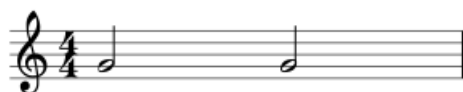


Figura 6 – Exemplo de Primeira Justa

A quarta justa (4^aJ 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 (5^{a}J ou $\text{P}5$) 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 (8^{a}J ou $\text{P}8$) ocorre quando há oito notas entre elas e a diferença é de cinco tons e dois semitons (Figura 9).

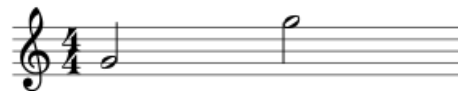


Figura 9 – Exemplo de Oitava Justa

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 (2^{a}m ou $\text{m}2$) ocorre quando a diferença é de um semitom, já a segunda maior (2^{a}M ou $\text{M}2$) 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 (3^{a}m ou $\text{m}3$) ocorre quando a diferença é de um tom e um semitom, já a terça maior (3^{a}M ou $\text{M}3$) 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 (6^am ou m6) ocorre quando a diferença é de três tons e dois semitons, já a sexta maior (6^aM 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 (7^am ou m7) ocorre quando a diferença é de quatro tons e dois semitons, já a sétima maior (7^aM 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 supraumentado, 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 (4^aA ou $A4$), quarta supraumentada (4^aSA ou $SA4^2$) e quarta 3x aumentada (4^a3xA 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 diminuto, 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 (6^ad ou $d6$), sexta supraumentada (6^asd ou $sd6^4$) e sexta 3x aumentada (6^a3xd 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 supraumentada 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 15^a corresponde a uma 1^a .

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 10^a

² Notação utilizada no código para intervalos supraumentados

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 (1996), 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 Ab, 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 diatônicas 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, segunda, terceira e quarta espécie (Figuras 23, 24, 25 e 27).

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 consonantes;
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 (10ªM);

8. só são permitidos, no máximo, quatro intervalos seguidos de terças ou sextas (maiores ou menores) seguidos;
9. se o movimento for paralelo, os intervallos 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. deve haver duas notas no contraponto para cada nota do *cantus firmus*;
2. deve haver uma nota no contraponto para a última nota do *cantus firmus*;
3. dissonâncias podem ser utilizadas em *arsis* (porções não-acentuadas do compasso);
4. 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;
5. o uníssono só pode ser utilizado no início, final ou em *arsis*;
6. se o uníssono for alcançado por salto (intervalo maior que uma segunda), deve ser deixado em grau conjunto na direção oposta à do salto;
7. evitar quintas e oitavas em *thesis* sucessivas, por se aproximarem de quintas e oitavas paralelas.



Figura 24 – Exemplo de Contraponto de Segunda Espécie

1.1.4.3 Contraponto de Terceira Espécie

Segundo [Jeppesen \(1992\)](#), o contraponto de segunda espécie possui, além das regras dos contrapontos de primeira e segunda espécie, as seguintes regras:

1. deve haver quatro notas no contraponto para cada nota do *cantus firmus*;
2. deve haver uma nota no contraponto para a última nota do *cantus firmus*;
3. dissonâncias podem ser utilizadas em *arsis* (porções não-acentuadas do compasso);
4. as dissonâncias podem ser utilizadas como nota de passagem ou como notas auxiliares (bordaduras) inferiores – bordaduras são alcançadas e deixadas em grau conjunto, mas em direções opostas;
5. As dissonâncias também podem ser empregadas por meio de *cambiata*s - sequência de cinco notas com intervalos de segunda descendente, terça ascendente e segunda ascendente - que iniciem em semínimas acentuadas (Figura 26);



Figura 25 – Exemplo de Contraponto de Terceira Espécie



Figura 26 – Exemplo de Cambiata

1.1.4.4 Contraponto de Quarta Espécie

Segundo Jeppesen (1992), o contraponto de quarta espécie possui, além das regras dos contrapontos citados anteriormente, as seguintes regras:

1. deve haver duas notas no contraponto para cada nota do *cantus firmus*;
2. dissonâncias devem ser utilizadas em *thesis* (porções acentuadas do compasso) e vir ligadas das *arsis* precedentes (síncope);
3. consonâncias devem ser utilizadas em *arsis* e resolver dissonâncias precedentes;
4. as dissonâncias devem ser resolvidas seguindo as seguintes regras: se o contraponto for superior, devem ser utilizadas quartas e sétimas resolvidas em terças e sextas, respectivamente, se ele for inferior, devem ser utilizadas segundas e nonas resolvidas em terças e décimas, respectivamente;
5. o uníssono pode ser utilizado livremente se estiver em cadeia de síncope, se a cadeia for quebrada, deve seguir as regras previstas na segunda espécie;



Figura 27 – Exemplo de Contraponto de Quarta 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 dado vértice).

Na Figura 28 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.

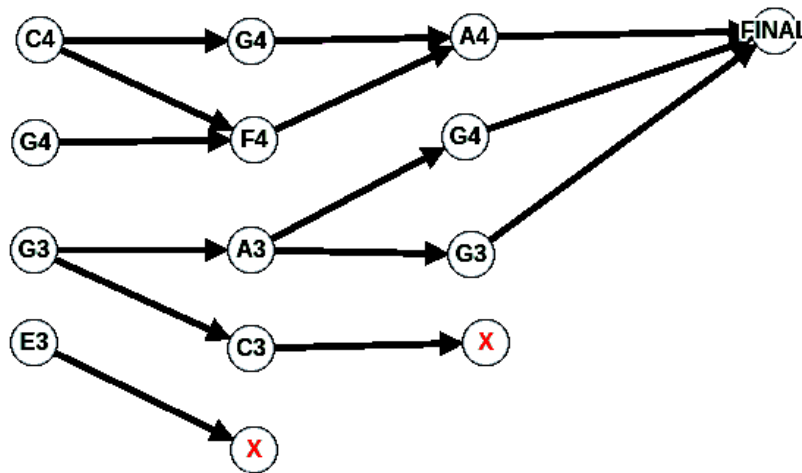


Figura 28 – Exemplo de Notas Possíveis Representados em Grafo

1.2.1 Grafos Direcionados Acíclicos

Um grafo pode possuir diversas classificações, sendo uma delas a classificação clássica é quanto à direção de suas arestas. 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 Laaksonen (2018). Veja a Figura 29.

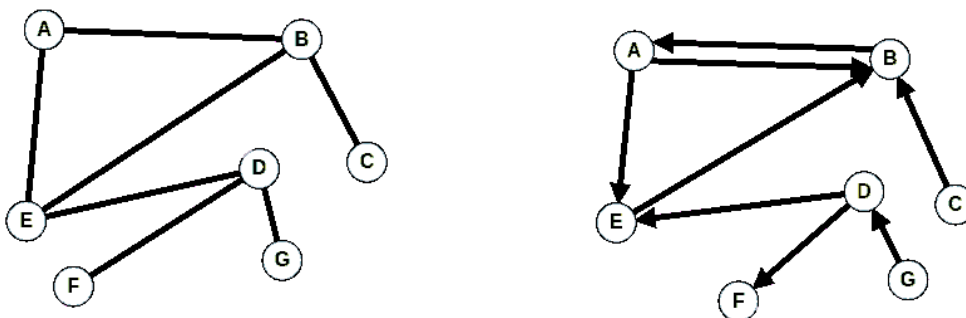


Figura 29 – Exemplos de Grafo Não-Direcionado e Direcionado

Grafos direcionados podem ser classificados como acíclicos ou não. 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 não acíclico.

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 30 exemplifica esses três tipos de grafos.

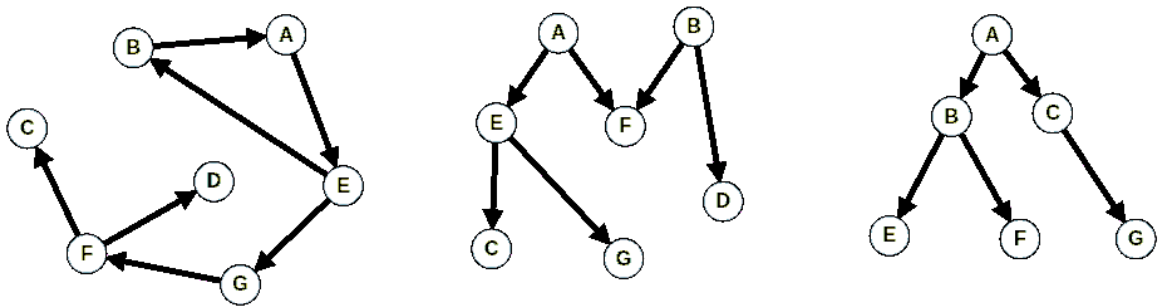


Figura 30 – Exemplos de Grafo Não-DAG, DAG e Árvore

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 suas 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 31.

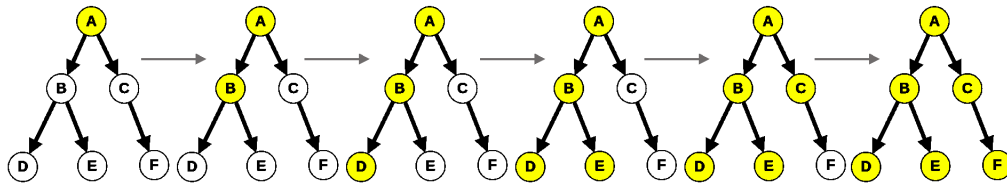


Figura 31 – Exemplo de DFS

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 32.

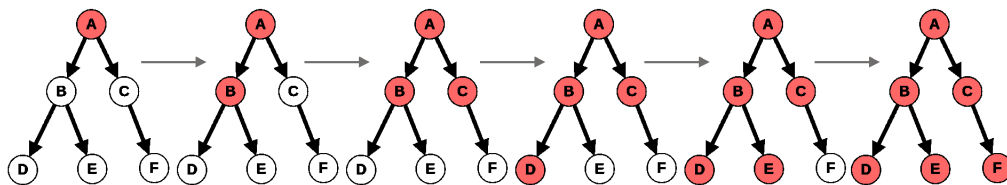


Figura 32 – Exemplo de BFS

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. Esse paradigma é focado na completude e não necessariamente na otimização do processo de busca de uma solução. Soluções implementadas utilizando tal paradigma percorrem todas as possibilidades de solução a fim de encontrar uma solução específica ou a melhor solução de acordo com critérios estabelecidos. Algoritmos de Busca Completa possuem alta complexidade ciclômática se comparados com algoritmos de Programação Dinâmica, por exemplo.

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

```
1  // Vector with 88 musicais notes found on piano
2  vector<Note> available_notes(88);
3
4  void backtracking(vector<Note> counterpoint) {
5      // Process solution if
6      // counterpoint candidate and cantus firmus
7      // has the same number of notes
8      if(counterpoint.size() == CANTUS_FIRMUS_SIZE){
9          process_solution(counterpoint);
10     }
11
12     // Append each available note to counterpoint
13     for(int i = 0; i < (int) available_notes.size(); i++){
14         counterpoint.push_back(available_notes[i]);
15         backtracking(counterpoint);
16         counterpoint.pop_back();
17     }
18 }
19
```

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 [Cormen \(2001\)](#). 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:

$$F(n) = \begin{cases} n & \text{se } n \leq 1 \\ F(n-1) + F(n-2) & \text{caso contrário} \end{cases}$$

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 33 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.

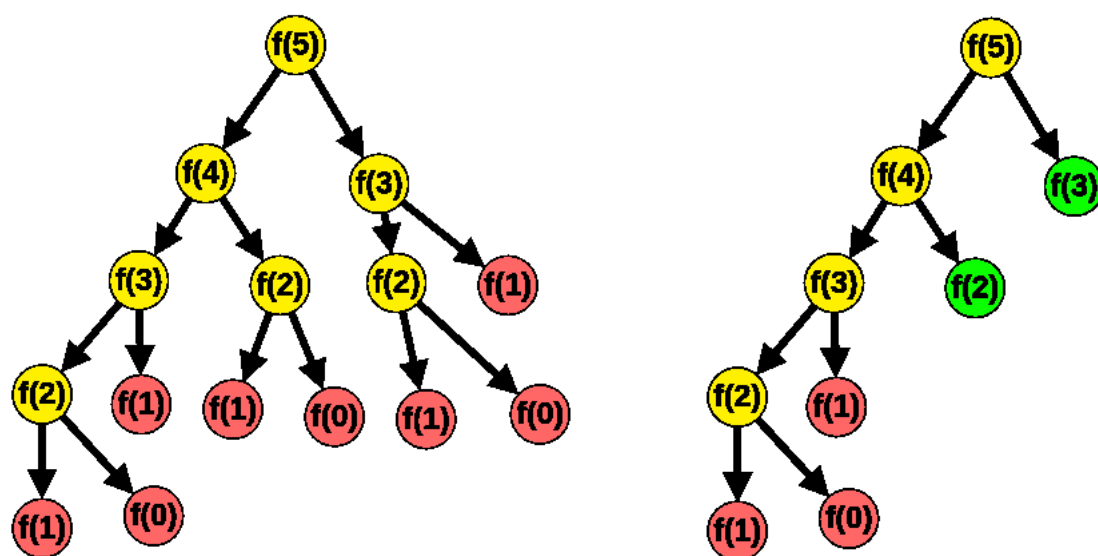


Figura 33 – Comparativo entre a abordagem não-DP e DP para o cálculo do $F(5)$

2 Metodologia

A implementação do Pierluigi utilizou 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 foi definido o escopo e as ferramentas a serem utilizadas. Então houve 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 foram as ferramentas utilizadas:

1. GitHub¹: plataforma para hospedagem de código-fonte que utiliza o Git como sistema de controle de versão. Utilizado para o armazenamento do código da solução pelo seu caráter aberto e gratuito;
2. C++², linguagem de programação compilada multiparadigma. Utilizada como linguagem de implementação principal pelo suporte à programação orientada a objetos e bibliotecas de leitura de arquivos e testes;
3. Ruby On Rails³, *framework* para aplicações *web*. Utilizado para desenvolvimento de uma camada *front-end* para a aplicação;
4. Lilypond⁴, formato de escrita musical baseado em TeX. 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;

¹ <<https://github.com/>>

² <<http://www.cplusplus.com/>>

³ <<https://rubyonrails.org/>>

⁴ <<http://lilypond.org/>>

5. MuseScore⁵, editor de partituras. 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.

2.2 Ciclo de Vida de Desenvolvimento

O ciclo de vida de desenvolvimento contou com três fases bem definidas. Primeiramente foram levantados os requisitos e então escolhidas as ferramentas a serem utilizadas. Após isso, foi desenvolvido um protótipo focado em contrapontos de primeira espécie. Os resultados do protótipo guiaram o desenvolvimento posterior, que foi focado na composição algorítmica de contrapontos palestrinianos de até quarta espécie e no módulo de construção de MIDI⁶.

2.2.1 Requisitos

O escopo inicial era de um *software* capaz de ler uma melodia monofônica e gerar contrapontos palestrinianos de até quarta espécie para ela no formato Lilypond. Veja um fluxograma simplificado da aplicação na Figura 34.

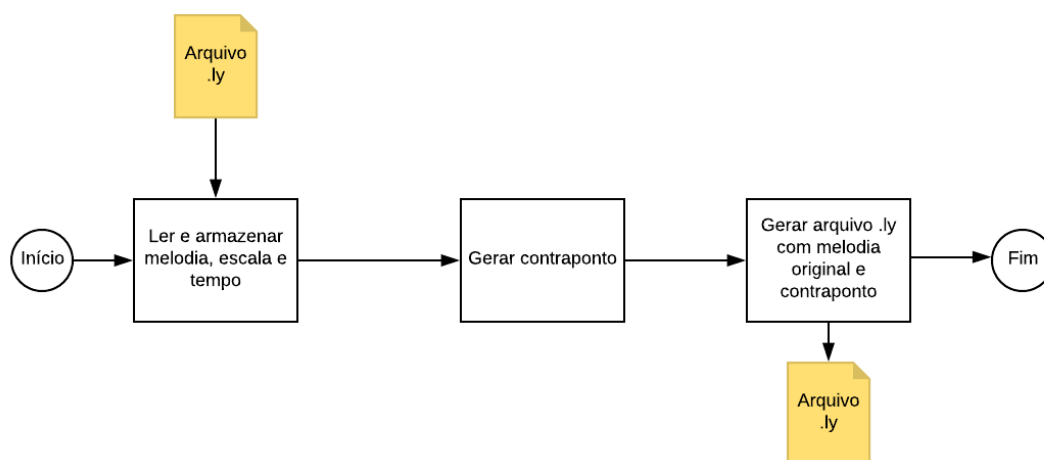


Figura 34 – Fluxograma simplificado da aplicação

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;

⁵ <<https://musescore.com/>>

⁶ *Musical Instrument Digital Interface* - Interface Digital de Instrumentos Musicais

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 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 Db.

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 arquivo lido deve ter as seguintes características para que funcione corretamente na solução:

1. Deve ter apenas uma melodia e ela deve ser monofônica. Se possuir mais de uma melodia, apenas a primeira será considerada para a geração do contraponto.
2. A melodia principal deve possuir escala e ritmo definidos explicitamente.
3. A nota mais curta da melodia principal deve ser uma colcheia sem pontos de aumento.

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 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 \flat , 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.

No desenvolvimento desse módulo é 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.

A DP é utilizada para a geração do contraponto. O estado é 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.

2.2.1.5 Módulo de Construção do MIDI

Segundo (GEORGE, 1996), o MIDI é um padrão de comunicação entre instrumentos musicais digitais amplamente aceito com especificações para *hardware* de comunicação, taxa de transferência de dados e formato de dados transferidos. Os arquivos .mid não são áudios, mas instruções digitais sobre como executar uma peça musical.

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

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 possui 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 teve o restante de suas funcionalidades implementadas e testadas. As atividades da 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.

O desenvolvimento iniciou 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, houve a otimização do *parser*. O *parser* do protótipo lê apenas um arquivo Lilypond modificado para conter apenas o tempo, a escala e as notas. Foi implementado um *parser* que, ainda que simples, é capaz de ler arquivos Lilypond sem tais modificações.

Seguindo a otimização do *parser*, houve a implementação e teste dos contrapontos de terceira e quarta espécie. Baseado no código já implementado dos contrapontos de

primeira e segunda espécie, eles foram desenvolvidos sequencialmente, já que as novas regras de uma espécie somam-se às regras da espécie anterior.

Por fim, houve a implementação de um módulo de construção de MIDI com as especificações definidas na Subseção 2.2.1.

2.3 Atividades

O desenvolvimento do sistema teve 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 construção de MIDI;
21. escrita do TCC 2.

3 Resultados

Primeiramente, 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. Na fase de desenvolvimento, foram implementados os testes, o módulo de construção de MIDI e o *parser* foi otimizado. A execução dos testes automatizados foi feita utilizando a biblioteca Catch2¹. A geração de contrapontos foi feita em uma interface desenvolvida em *Ruby On Rails*.

3.1 Módulo de Notação Musical

O módulo de notação musical possui quatro classes: *Note*, *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.

¹ <<https://github.com/catchorg/Catch2C++>>

8. *Valid*: atributo booleano que indica se a nota é válida ou não, útil para retornar notas inválidas sem utilizar ponteiros, quando necessário.
9. *Linked*: atributo booleano que indica se a nota está ou não ligada à anterior.

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 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.

A Figura 35 descreve os atributos, métodos e construtores da classe.

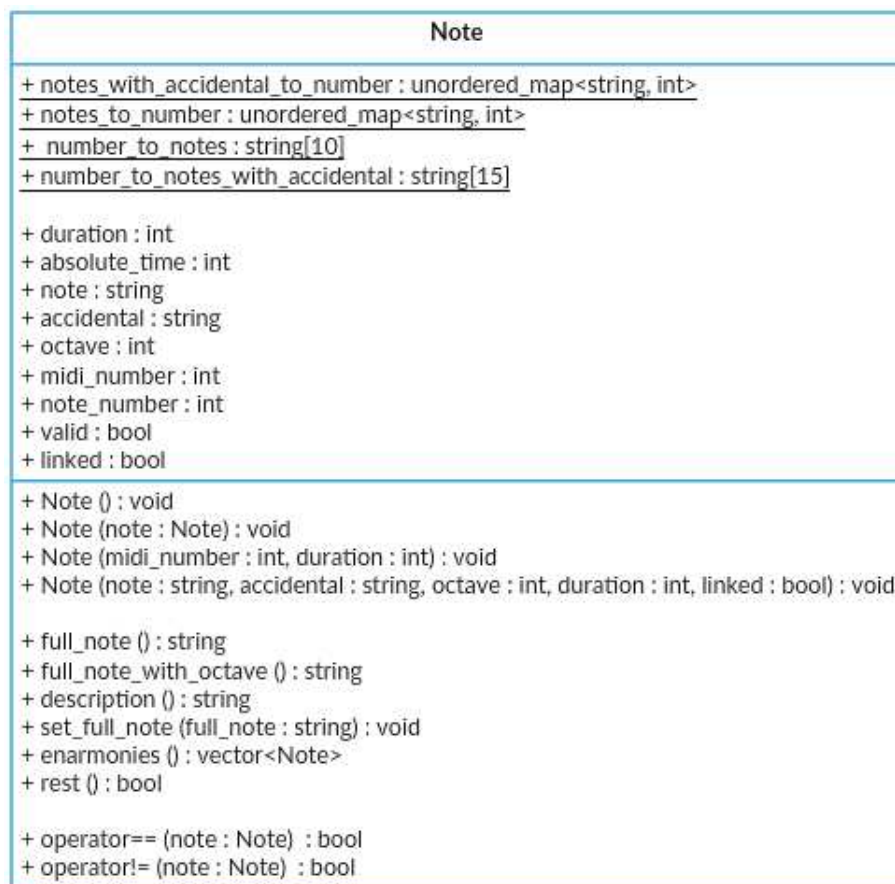


Figura 35 – Diagrama da Classe *Note*

3.1.2 *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 36 representa os atributos e métodos da classe.

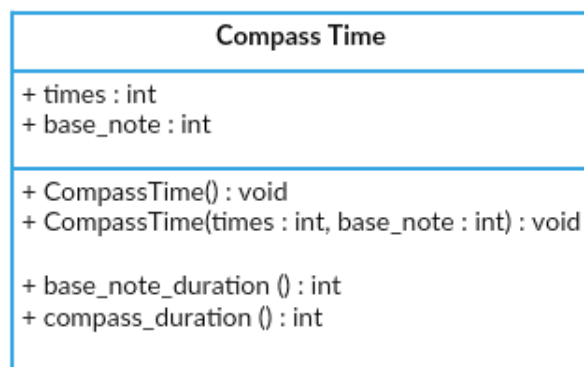


Figura 36 – Diagrama da Classe *Compass Time*

3.1.3 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.

Em relação aos métodos, essa classe possui os seguintes:

1. *Size*: Retorna o número de notas da melodia.
2. *Back*: Retorna a última nota da melodia ou uma nota inválida se a melodia não tiver nenhuma nota no estado atual.
3. *Size Without Rest*: Retorna o número de notas da melodia sem notas de descanso que possam haver ao final desta.
4. *Trailing Rests*: Retorna o número de descansos ao final da melodia.

A Figura 37 representa os atributos e métodos da classe.

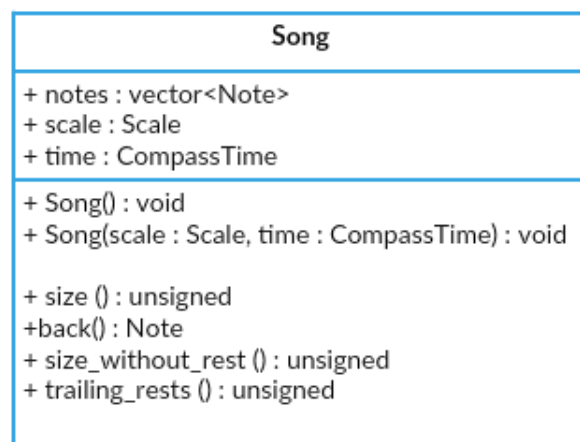


Figura 37 – Diagrama da Classe *Song*

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. *Direction*: Inteiro que indica se o intervalo é ascendente, descendente ou uníssono, 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. *Ascendant*: retorna um booleano definindo se o intervalo é ou não ascendente, definido a partir da primeira nota em relação à segunda.
4. *Descendant*: retorna um booleano definindo se o intervalo é ou não descendente.
5. *Ascendant Or Unison*: retorna um booleano definindo se o intervalo é ou não ascendente ou uníssono.
6. *Descendant Or Unison*: retorna um booleano definindo se o intervalo é ou não descendente ou uníssono.
7. *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 38 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*.

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 39 representa os atributos e métodos da classe.

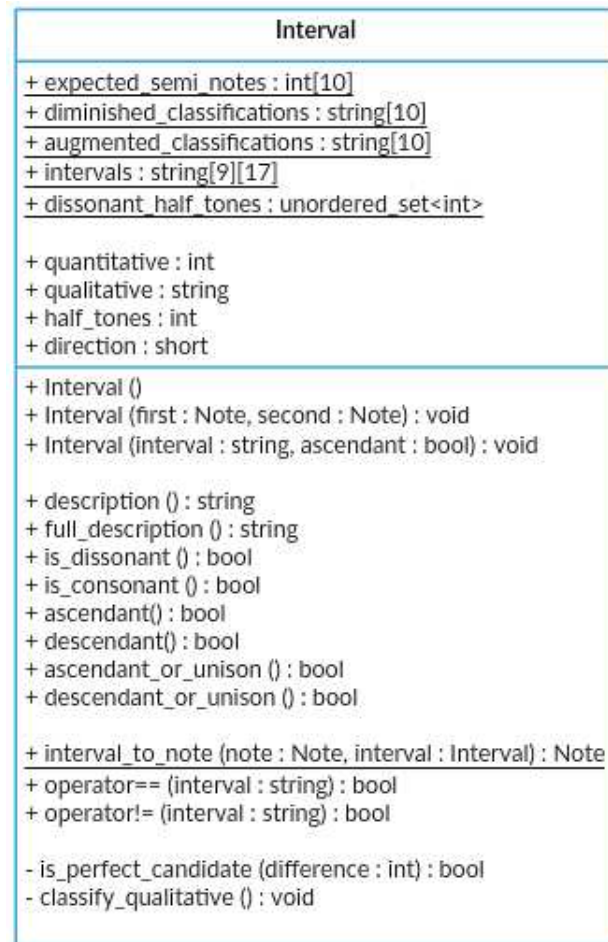


Figura 38 – Diagrama da Classe *Interval*

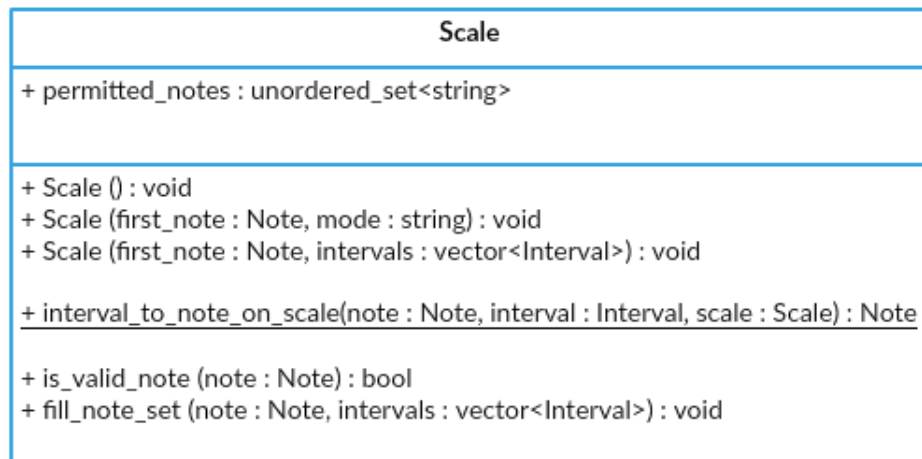
3.4 Módulo de Contraponto

O módulo de contraponto é responsável pela geração de contrapontos. Utilizando todas as classes anteriormente citadas, possui cinco classes: *Counterpoint*, *First Species Counterpoint*, *Second Species Counterpoint*, *Third Species Counterpoint* e *Fourth Species Counterpoint*.

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 espécie, atualmente possui atributos estáticos como a matriz *DP* utilizada para armazenar estados da solução e um ponteiro para *Song* utilizado para guardar referência para o *cantus firmus* durante toda a solução, possui ainda o método ad-hoc de geração de contrapontos e o método *Analyse and Add Interval*.

Além disso, possui o método *Add Trailing Rests* que adiciona descanso ao final do contraponto que não foram utilizados ao gerar a solução final. Esse método é importante

Figura 39 – Diagrama da Classe *Scale*

para os algoritmos de segunda e terceira espécie, em que a última nota que não é descanso possui apenas uma nota no mesmo tempo no contraponto. O método *Is Thesis* diz se uma nota está ou não no tempo forte do compasso dada sua posição, utilizado para utilizar ou não dissonâncias em dados momentos da solução nos algoritmos de segunda a quarta espécie.

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.

Essa classe é utilizada como classe pai das classes dos contrapontos de primeira a quarta espécie. A Figura 40 apresenta os atributos e métodos da classe.

3.4.2 First Species Counterpoint

A classe *First Species Counterpoint* herda da classe *Counterpoint* e é uma biblioteca, possuindo métodos de classe. Ela possui os seguintes métodos:

1. *DFS Generate Counterpoint*: recebe uma instância de *Song*, um booleano que define se o contraponto será superior ou inferior e os números de terças e sextas paralelas

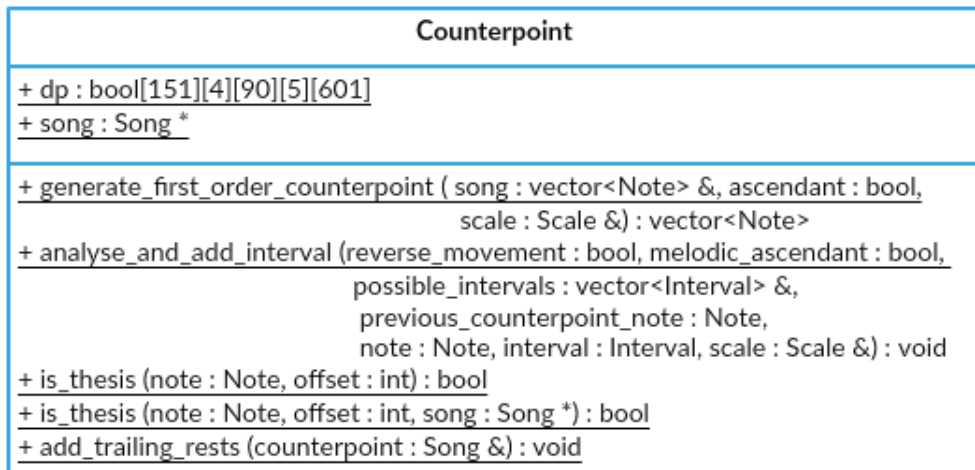


Figura 40 – Diagrama da Classe *Counterpoint*

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.
3. *Analyse And Add Interval*: método de análise de notas para inserção no contraponto, avalia regras como tamanho do salto e natureza do movimento (reverso ou não) e adiciona o intervalo sugerido na lista de possíveis intervalos caso ele atenda às regras estabelecidas.

A Figura 41 apresenta os métodos e atributos da classe.

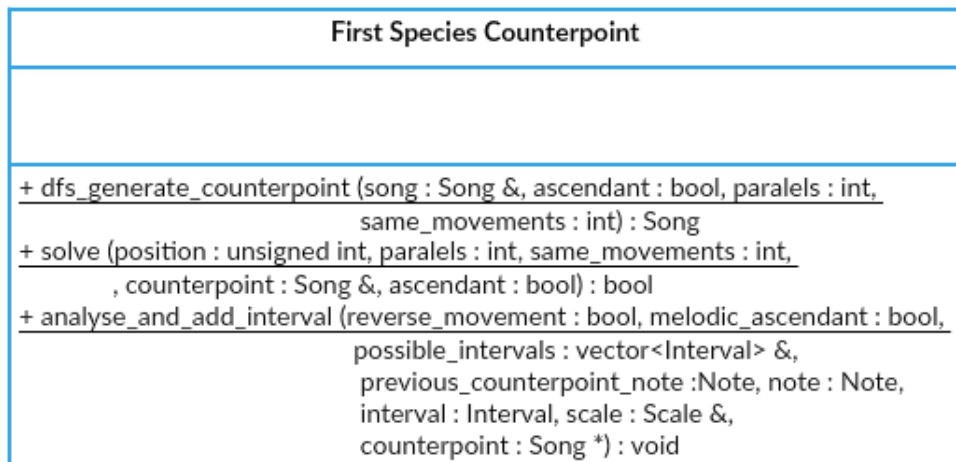


Figura 41 – Diagrama da Classe *First Species Counterpoint*

3.4.3 *Second Species Counterpoint*

A classe *Second Species Counterpoint* herda da classe *Counterpoint* e possui os mesmos métodos apresentados em 3.4.2. A Figura 42 apresenta os métodos e atributos da classe.

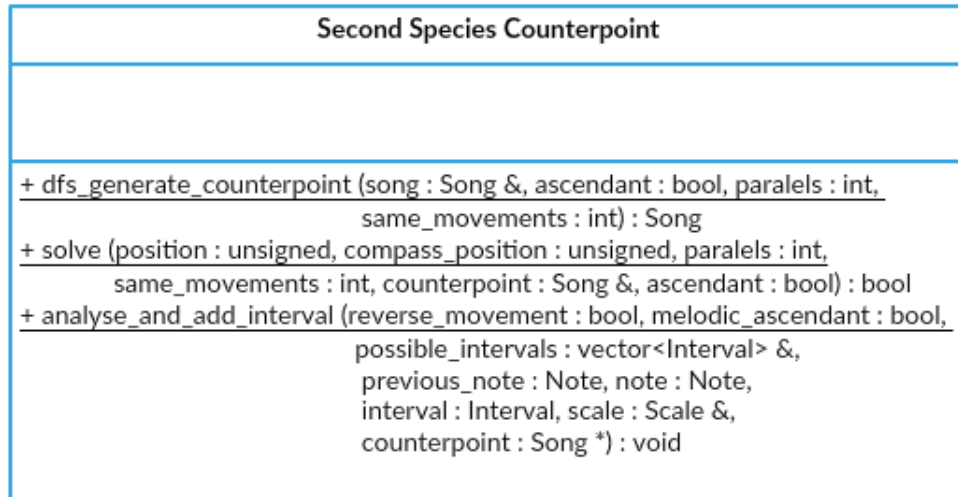


Figura 42 – Diagrama da Classe *Second Species Counterpoint*

3.4.4 *Third Species Counterpoint*

A classe *Third Species Counterpoint* herda da classe *Counterpoint* e possui os mesmos métodos apresentados em 3.4.2. A Figura 43 apresenta os métodos e atributos da classe.

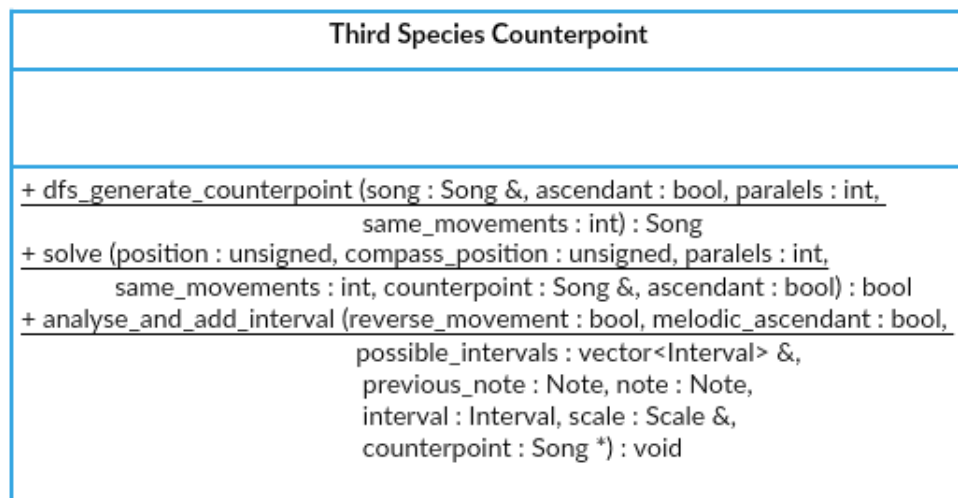


Figura 43 – Diagrama da Classe *Third Species Counterpoint*

3.4.5 *Fourth Species Counterpoint*

A classe *Fourth Species Counterpoint* herda da classe *Counterpoint* e possui os mesmos métodos apresentados em 3.4.2. A Figura 44 apresenta os métodos e atributos da classe.

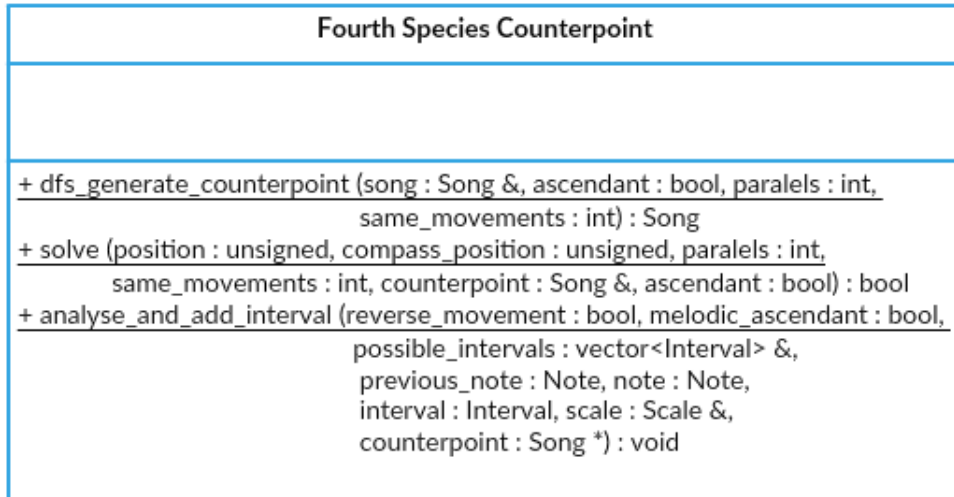


Figura 44 – Diagrama da Classe *Fourth Species Counterpoint*

O método que retorna o vetor de contrapontos encapsula a solução e chama o método recursivo. O algoritmo possui o seguinte funcionamento: iniciando na primeira posição do *cantus firmus*, o estado da DP é definido por cinco parâmetros: posição, posição no compasso, 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 booleano de cinco dimensões, uma para cada estado. A primeira possui 151 espaços, permitindo que um contraponto seja gerado para um *cantus firmus* com, no máximo, 151 notas. A segunda dimensão possui 4 espaços e indica a posição do contraponto em relação à nota atual, especialmente utilizado em contrapontos de segunda a quarta espécie, em que cada nota do *cantus firmus* possui duas ou quatro notas no tempo equivalente no contraponto. Em contrapontos de primeira espécie, esse valor é sempre 0, em contrapontos de segunda e quarta espécie vai de 0 a 1 e em contrapontos de terceira espécie vai de 0 a 4. A terceira dimensão possui 90 espaços e representa o número MIDI da nota atual e cobre as 88 primeiras notas do piano. A quarta dimensão possui 5 espaços e cobre as 4 terças ou sextas paralelas seguidas permitidas pelo algoritmo. A quinta dimensão representa o número de movimentos paralelos que aconteceram durante o contraponto, possui 601 espaços pois o máximo de movimentos paralelos permitidos pelo algoritmo é a música completa multiplicada por 4, que é o máximo número de notas em um contraponto de terceira espécie.

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 terceira nota do contraponto para aquela nota do *cantus firmus*, 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][2][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 45 apresenta o fluxograma do algoritmo.

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

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. *Scale To String*: retorna uma instância da *Scale* ao receber uma *string*.
4. *String To Scale*: retorna uma *string* que representa escalas no formato Lilypond ao receber um objeto da classe *Scale*.
5. *Compass Time To String*: retorna uma instância da *CompassTime* ao receber uma *string*.
6. *String To Compass Time*: retorna uma *string* que representa ritmo no formato Lilypond ao receber um objeto da classe *CompassTime*.
7. *Song Clef To String*: retorna uma *string* que define em qual clave a melodia deve ser impressa baseada no valor médio da nota – uma melodia com notas mais agudas é impressa na clave de sol e uma melodia com notas mais baixas é impressa na classe de fá.
8. *Song To Voice String*: retorna uma *string* que representa a voz do contraponto no Lilypond, a voz é impressa com clave, escala e ritmo seguidos das notas musicais.
9. *New Staff String*: retorna uma *string* que define uma nova voz no arquivo Lilypond, sendo essa voz nomeada de *CounterpointVoice*.
10. *Midi Indicator*: retorna uma *string* que indica que o lilypond em que ela for adicionada deve gerar um arquivo MIDI além do PDF da partitura.
11. *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.
12. *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 46 representa os métodos da classe.

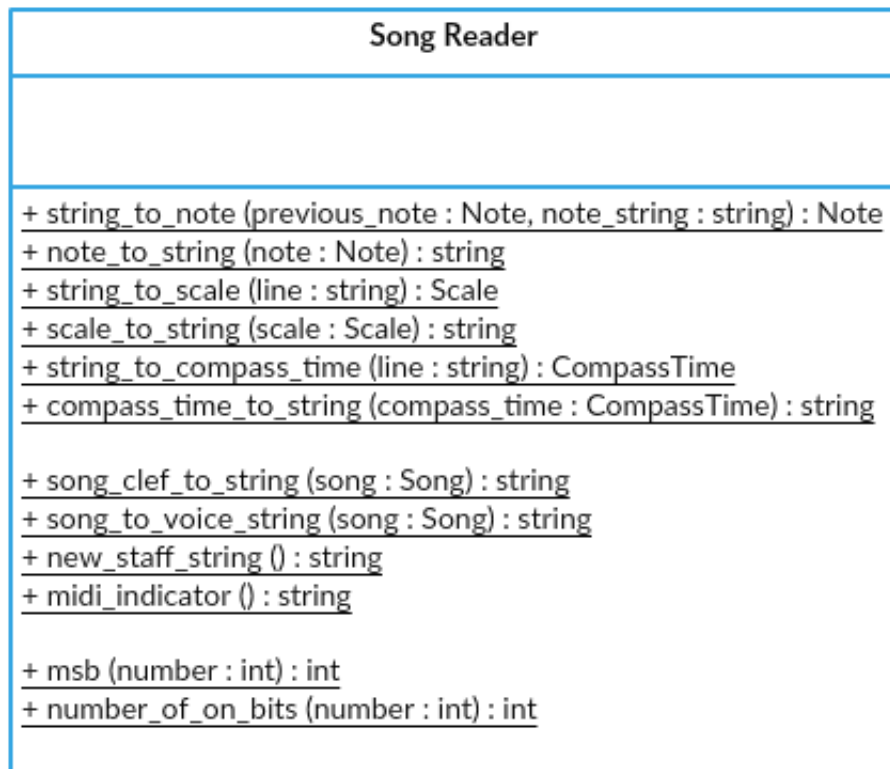


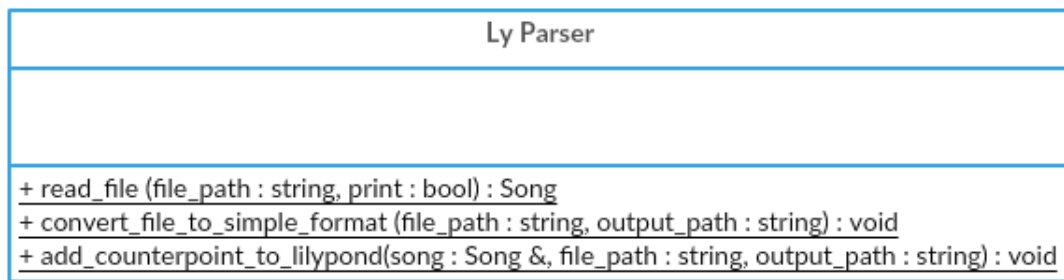
Figura 46 – Diagrama da Classe *Song Reader*

3.6.2 *Ly Parser*

A classe *Ly Parser* é responsável pela leitura de um arquivo Lilypond completo e a geração de um novo arquivo Lilypond completo com o contraponto inserido. Para isso, ela primeiramente lê o arquivo original e gera um arquivo simplificado a partir dele. Após isso, ela lê os dados do arquivo simplificado utilizando a classe *Song Reader*. Após a geração do contraponto, ela adiciona o contraponto ao arquivo original, gerando um novo arquivo Lilypond. Ela possui os seguintes métodos:

1. *Read File*: lê um arquivo Lilypond simplificado em formato pré-estabelecido e retorna uma instância da classe *Song* com escala, ritmo e notas lidas.
2. *Convert File To Simple Format*: Recebe o caminho de um arquivo Lilypond completo e gera um arquivo Lilypond simplificado em outro caminho especificado.
3. *Add Counterpoint to Lilypond*: Recebe um contraponto gerado, o caminho do arquivo original e gera um novo arquivo Lilypond com a voz do contraponto gerado adicionada.

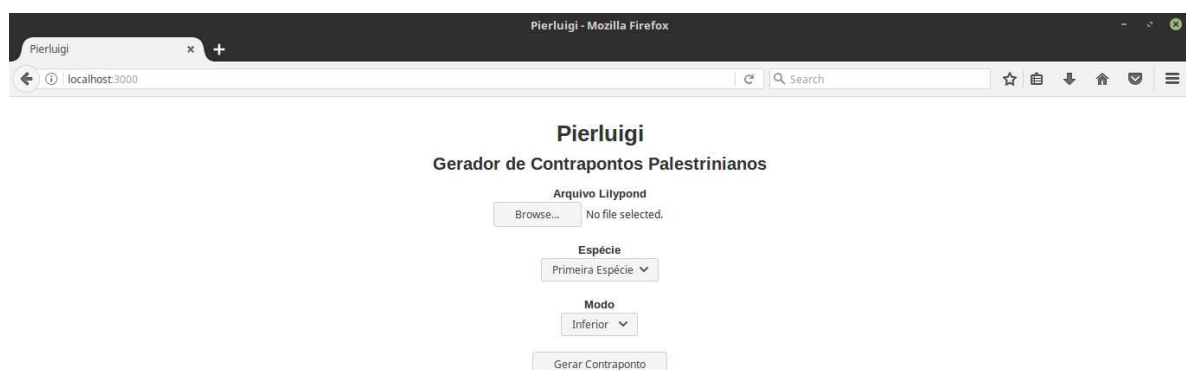
A Figura 47 apresenta os métodos e atributos da classe.

Figura 47 – Diagrama da Classe *Ly Parser*

3.6.3 Counterpoint Generator

O *Counterpoint Generator* é uma aplicação *web* simples, desenvolvida em *Ruby On Rails*. Nela, seleciona-se o arquivo Lilypond escolhido para se gerar um contraponto, a espécie de contraponto e se ele será inferior ou superior. Utilizando o executável do algoritmo em C++, o contraponto é gerado e o novo arquivo Lilypond é utilizado para gerar a partitura em PDF e o arquivo .mid. A aplicação compacta o arquivo Lilypond, o PDF e o MIDI e os envia para o usuário.

A Figura 48 apresenta o acesso ao *front-end* da aplicação.

Figura 48 – Página *web* do *Pierluigi*

3.7 Experimentos

Para validar o 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/pierluigi>, o código-fonte do *front-end* da aplicação está disponível em https://github.com/joao18araujo/pierluigi_frontend.

Todos os módulos implementados possuem testes automatizados. Para isso, utilizou-se a biblioteca Catch2. Para garantir a corretude dos testes durante o desenvolvimento,

foi utilizada uma ferramenta de integração contínua no repositório. O resultado dos testes pode ser observado em <https://travis-ci.org/joao18araujo/pierluigi>.

A música original pode ser observada em formato Lilypond e em partitura no Código 3.1 e na Figura 49, respectivamente.

Código 3.1 – *Twinkle Twinkle Little Star*

```

1  \key c \major
2  \time 4/4
3  c''4 c'' g'' g''
4  a'' a'' g''2
5  f''4 f'' e'' e''
6  d'' d'' c''2
7  g''4 g'' f'' f''
8  e'' e'' d''2
9  g''4 g'' f'' f''
10 e'' e'' d''2
11 c''4 c'' g'' g''
12 a'' a'' g''2
13 f''4 f'' e'' e''
14 d'' d'' c''2
15

```



Figura 49 – Partitura de *Twinkle Twinkle Little Star*

Após a execução do algoritmo para contraponto de primeira espécie, o contraponto gerado pode ser visto em partitura na Figura 50.

Após a execução do algoritmo para contraponto de segunda espécie, o contraponto gerado pode ser visto em partitura na Figura 51.

Após a execução do algoritmo para contraponto de terceira espécie, o contraponto gerado pode ser visto em partitura na Figura 52.

Após a execução do algoritmo para contraponto de quarta espécie, o contraponto gerado pode ser visto em partitura na Figura 53.

Melodia

Contraponto

7

Figura 50 – Partitura do contraponto de primeira espécie gerado

Melodia

Contraponto

5

Ctp.

9

Ctp.

Figura 51 – Partitura do contraponto de segunda espécie gerado

4 Considerações Finais

O principal desafio do trabalho foi a implementação de regras mais complexas dos contrapontos de segunda espécie em diante. A solução prevista no protótipo considerava cada nota como um elemento individual, sem considerar o compasso em que está inserida. Tal decisão teve que ser revista na implementação das outras espécies de contraponto, pois era necessário saber se a nota estava em tempo forte ou fraco, algo definido pela posição em que a nota ocupa no compasso.

O Lilypond provou ser um bom formato para se trabalhar com música programaticamente, os módulos de leitura de arquivo e construção de MIDI tiveram o desenvolvimento facilitado pelo modo compacto e não redundante que os arquivos .ly possuem. Contudo, devido à forma como vários formatos podem ser convertidos para Lilypond, não há garantia total de que o *parser* implementado funciona com qualquer arquivo Lilypond que siga as regras pré-estabelecidas. Os testes conduzidos utilizando arquivos Lilypond gerados a partir de arquivos *MusicXML* foram bem-sucedidos.

A solução utilizando DP possui limitações relacionadas ao número de estados possíveis. Quanto maior a espécie do contraponto, mais regras são aplicadas e nem todas podem ter a decisão relacionada a ela representada como um estado da DP por falta de memória.

Há regras que tiveram que ser aplicadas, mas não foram representadas em estado. Tal decisão cria a possibilidade de que possam ser devolvidas respostas indicando que não há solução, quando haveria se a ordem de aplicação das regras fosse diferente. A aleatoriedade do algoritmo pode ser utilizada para executar o algoritmo diversas vezes, retornando uma solução válida não considerada em iterações anteriores. Vale ressaltar que tal caso não ocorreu em nenhum dos testes executados. A possibilidade de acontecer é relativamente baixa pois a medida que os contrapontos aumentam em número de regras, mais contrapontos válidos podem ser gerados, mitigando esse problema.

A implementação de testes automatizados provou ser uma ótima ferramenta de detecção de *bugs*. Sua construção teria sido ainda mais útil se construída junto com o protótipo, pois diminuiria a quantidade de *bugs* simples que acabaram tendo resoluções mais complexas pelo fato de terem sido descobertos na fase de desenvolvimento.

A partir de regras bem definidas e a representação de música de uma forma digital, foi possível gerar contrapontos palestrinianos de primeira a quarta espécie algoritmicamente. Com isso, pode-se afirmar que o trabalho foi concluído com sucesso. Os módulos de notação musical, escala, intervalo, contraponto e de construção de MIDI estão completos, funcionais e testados.

4.1 Trabalhos Futuros

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

1. Resolver *bugs* da implementação incompleta do contraponto de terceira e quarta espécie, eles já são gerados, mas ferem algumas regras mais complexas.
2. Refatorar as classes já implementadas para aumentar a solidez do código.
3. Implementar geração de contrapontos para melodias polifônicas.
4. Melhorar interface do *front-end* da aplicação, que atualmente possui caráter básico.
5. Implementar os módulo de contrapontos livres e de quinta espécie.

Referências

CORMEN, T. *Introduction to algorithms*. Cambridge, Mass: MIT Press, 2001. ISBN 0070131511. Citado na página 37.

GEORGE, K. MIDI Inline Transposer. *Proceedings of the 39th Midwest Symposium on Circuits and Systems*, 21 Aug. 1996, p. 1, 1996. Citado na página 43.

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 19.

GOGAL, M.; GOGA, N. Feelings Based Computer Music. p. 673–676, 2004. ISSN 08407789. Citado na página 19.

HALIM, S. *Competitive Programming 3*. Singapura, 2013. 447 p. Citado 2 vezes nas páginas 33 e 36.

JEPPESEN, K. *Counterpoint: the polyphonic vocal style of the Sixteenth Century*. Nova York, Estados Unidos da América, 1992. 320 p. Citado 4 vezes nas páginas 24, 30, 31 e 32.

KOELLREUTER, H.; JOACHIM, H. *Contraponto Modal do Século XVI (Palestrina)*. Brasília, Brasil, 1996. 85 p. Citado na página 30.

LAAKSONEN, A. *Competitive Programmer's Handbook*. Finland, 2018. 296 p. Citado na página 34.

MED, B. *Teoria da Música*. Brasília, Brasil, 1996. 420 p. Citado 3 vezes nas páginas 21, 25 e 28.

TELES, V. M. *Extreme Programming*. Rio de Janeiro, Brasil, 2014. 328 p. Citado na página 39.

TRAGTENBERG, L. *Contraponto: Uma Arte de Compor*. São Paulo, Brasil, 1961. 266 p. Citado na página 19.