

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CIÊNCIA DA COMPUTAÇÃO

Sergio Bonini

Geração de Solos de Guitarra como um Problema de Otimização

Florianópolis
10 de dezembro de 2025

Sergio Bonini

Geração de Solos de Guitarra como um Problema de Otimização

Trabalho de Conclusão de Curso submetido ao Curso de Graduação em Ciência da Computação do Centro Tecnológico da Universidade Federal de Santa Catarina como requisito para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Pedro Belin Castellucci

Florianópolis
10 de dezembro de 2025

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Bonini, Sergio
Geração de Solos de Guitarra como um Problema de
Otimização / Sergio Bonini ; orientador, Pedro Belin
Castellucci, 2025.
50 p.

Trabalho de Conclusão de Curso (graduação) -
Universidade Federal de Santa Catarina, Centro Tecnológico,
Graduação em Ciências da Computação, Florianópolis, 2025.

Inclui referências.

1. Ciências da Computação. 2. Otimização . 3. Programação
Inteira. 4. Música. I. Castellucci, Pedro Belin. II.
Universidade Federal de Santa Catarina. Graduação em
Ciências da Computação. III. Título.

Sergio Bonini
Geração de Solos de Guitarra como um Problema de Otimização

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharel em Ciência da Computação e aprovado em sua forma final pelo curso de Graduação em Ciência da Computação.

Florianópolis, 10 de dezembro de 2025.

Banca Examinadora:

Prof. Dr. Pedro Belin Castellucci
Orientador
Universidade Federal de Santa Catarina

Prof. Dr. Rafael de Santiago
Avaliador
Universidade Federal de Santa Catarina

Profa. Dra. Aline Aparecida de Souza Leão
Avaliador
Universidade Estadual de Londrina

AGRADECIMENTOS

Agradeço ao meu orientador Pedro Belin Castellucci por toda a supervisão ao longo do trabalho e elaboração das cartas de recomendação para os programas de mestrado. Também agradeço aos professores Rafael de Santiago e Jerusa Marchi pela elaboração das cartas de recomendação. Agradeço à minha família pelo suporte ao longo da graduação.

RESUMO

Neste trabalho, a geração musical automática foi explorada, especificamente para a geração de solos de guitarra. A atividade foi tratada como um problema de otimização no qual um aglomerado de conjuntos de notas musicais foi agrupado a fim de se gerar a melhor combinação possível em termos de qualidade musical, baseando-se em opiniões previamente coletadas de músicos profissionais. Para tal objetivo, utilizou-se um artigo que contém o *dataset* com os conjuntos de notas musicais explorados, bem como os critérios para a atribuição de valores, que indicam a qualidade musical para cada transição entre estes conjuntos. Ainda com base no artigo, a formulação do problema de otimização em termos de programação inteira foi elaborada e utilizou-se a biblioteca PuLP da linguagem Python, com o *solver* CBC. Em seguida, uma análise do programa solução e seu desempenho foram realizados e soluções alternativas que podem ser avaliadas em trabalhos futuros foram citadas. Comparações entre os conjuntos de notas musicais gerados no experimento e no artigo base foram feitas, além do impacto de diferentes parâmetros no custo computacional para a solução do modelo. Por fim, observaram-se resultados sonoros divergentes dos obtidos no artigo de referência, mas positivos no quesito tempo de execução para a obtenção das soluções ótimas para todas as instâncias avaliadas, tornando a utilização do programa viável para instâncias de diferentes tamanhos, dentro dos limites de tamanho impostos pela dimensão do *dataset*.

Palavras-chave: Programação Inteira. Solos de Guitarra. Geração Musical.

ABSTRACT

In this work, automatic music generation was explored, specifically for the generation of guitar solos. The activity was treated as an optimization problem in which a cluster of musical note sets was grouped in order to generate the best possible combination in terms of musical quality, based on previously collected opinions from professional musicians. For this purpose, an article containing the dataset with the explored musical note sets was used, as well as the criteria for assigning values, which indicate the musical quality for each transition between these sets. Also based on the article, the formulation of the optimization problem in terms of integer programming was elaborated and the PuLP library of the Python language was used, with the CBC solver. Then, an analysis of the solution program and its performance was carried out and alternative solutions that can be evaluated in future works were cited. Comparisons between the musical note sets generated in the experiment and in the base article were made, in addition to the impact of different parameters on the computational cost for the model solution. Finally, the results observed differed from those obtained in the reference article, but were positive in terms of execution time for obtaining optimal solutions for all evaluated instances, making the use of the program viable for instances of different sizes, within the size limits imposed by the dataset dimension.

Keywords: Integer Programming. Guitar Solos. Music Generation.

LISTA DE FIGURAS

Figura 1 – Obtenção de uma solução ótima $\bar{z} = 16,38$ para o problema relaxado PL para $\bar{x}_1 = 3,83$ e $\bar{x}_2 = 2,75$. O vetor gradiente $\nabla z = (5, -1)$ aponta para a direção em que a função objetivo z cresce mais rapidamente. (Arenales et al., 2007)	25
Figura 2 – Problema inicial de programação inteira PI (ou P) é relaxado, resolvido e dividido em dois novos subproblemas de programação inteira PI_1 e PI_2 , cada um com novas restrições. (Arenales et al., 2007)	26
Figura 3 – PI_1 é relaxado linearmente, tornando-se PL_1 , que é resolvido, obtendo-se $\bar{x}_1 = 3$ e $\bar{x}_2 = 1,59$, com a função objetivo $\bar{z} = 13,07$. PI_2 é infactível. (Arenales et al., 2007)	26
Figura 4 – P_1 gera dois novos subproblemas P_3 e P_4 . (Arenales et al., 2007)	27
Figura 5 – P_3 e P_4 são relaxados linearmente em PL_3 e PL_4 , respectivamente, e a solução de PL_4 é adotada como solução para P . (Arenales et al., 2007)	27
Figura 6 – Árvore de soluções para o exemplo 2 gerada a partir do método <i>branch-and-bound</i> . (Arenales et al., 2007)	30
Figura 7 – Aplicação do método <i>branch-and-cut</i> para a resolução do exemplo 2, evidenciando uma menor quantidade de nodos na árvore de solução quando comparado ao método <i>branch-and-bound</i> para o mesmo problema. (Arenales et al., 2007)	31
Figura 8 – Um caminho contendo 10 nodos, mais os nodos inicial e final, de modo que, dentre todos os caminhos possíveis de 10 nodos, este seja um dos caminhos que apresentam o custo mínimo. Nodos brancos são regulares, listrados são de repetição, cinza são de <i>turnaround</i> e pretos são os que começam e/ou terminam com uma pausa. (Cunha et al., 2017).	38

LISTA DE TABELAS

Tabela 1	– Trabalhos relacionados em ordem cronológica	33
Tabela 2	– Com base na natureza de um par de <i>licks</i> L1 e L2 quaisquer, é definido o custo de transição de L1 para L2. Este custo de transição pode ser do tipo bônus ou penalidade e cada tipo de transição está associado a uma preferência T1, T2, ... T8. (Cunha et al., 2017)	36
Tabela 3	– Categorias que cada <i>lick</i> pode apresentar e as quantidades que cada categoria de <i>lick</i> está presente no <i>dataset</i> . (Cunha et al., 2017)	37
Tabela 4	– Médias e medianas dos valores das soluções ótimas encontradas (quanto menor, maior a qualidade), do tempo gasto pelo modelo para se atingir uma solução ótima e do número de subciclos encontrados antes de se determinar uma solução ótima. L representa o número de <i>licks</i> possíveis de serem escolhidos para compor a solução ótima (parâmetro b foi considerado 12 para todos estes casos de teste, ou seja, o número de compassos e, consequentemente, o número de <i>licks</i> que devem estar presentes em cada uma das soluções ótimas foi constante).	41
Tabela 5	– Médias e medianas do tempo de solução, número de subciclos e qualidade da solução ótima, para diferentes perfis de <i>licks</i> e valores de L (quanto menor o valor de qualidade, melhor o solo).	54

SUMÁRIO

1	INTRODUÇÃO	15
1.1	OBJETIVOS	16
2	FUNDAMENTAÇÃO TEÓRICA	17
2.1	CONCEITOS DE TEORIA MUSICAL	17
2.2	PROBLEMA DO CAIXEIRO VIAJANTE (TSP)	18
2.3	PROBLEMA DO CAMINHO MAIS CURTO ELEMENTAR COM RES- TRIÇÕES DE RECURSOS (ESPPRC)	19
2.4	MÉTODOS DE SOLUÇÃO	22
2.4.1	Preliminares de um problema de otimização	22
2.4.2	<i>Branch-and-bound</i>	23
2.4.3	Planos de corte	28
2.4.4	<i>Branch-and-cut</i>	29
3	TRABALHOS RELACIONADOS	32
4	DEFINIÇÃO DO PROBLEMA E PROPOSTA DE SOLUÇÃO	35
4.1	ARRANJOS DE NOTAS MUSICAIS	35
4.2	MODELAGEM COMPUTACIONAL PARA COMPOSIÇÃO DE SOLOS	35
4.2.1	Construção do grafo de <i>licks</i>	35
4.2.2	Formulação do problema de programação linear inteira-mista	38
4.2.3	Pós-processamento	40
5	RESULTADOS	41
5.1	<i>LICKS</i> DOS TIPOS <i>SLOW</i> , <i>MODERATE</i> E <i>FAST</i>	41
5.2	ANÁLISE QUALITATIVA	43
6	CONCLUSÃO	44
	REFERÊNCIAS	45
A	RESUMO EM FORMATO DE ARTIGO DA SBC	47
A.1	INTRODUÇÃO	49
A.2	TRABALHOS RELACIONADOS	50
A.3	DEFINIÇÃO DO PROBLEMA E ABORDAGEM PROPOSTA	51
A.3.1	Arranjos de notas musicais e <i>dataset</i>	51
A.3.2	Modelagem em grafo e formulação em PLIM	51
A.3.3	Fluxo de implementação e pós-processamento	53
A.4	RESULTADOS EXPERIMENTAIS	53

A.4.1	Discussão dos resultados numéricos	54
A.4.2	Análise qualitativa	55
A.5	CONCLUSÕES E TRABALHOS FUTUROS	55
B	CÓDIGO DESENVOLVIDO	57
C	EXEMPLOS DE ARQUIVOS DE SAÍDA	68

1 INTRODUÇÃO

A geração musical por meios computacionais tem se tornado um campo cada vez mais explorado, impulsionado tanto pelos avanços em inteligência artificial quanto pela crescente disponibilidade de dados musicais digitais (ZHU et al., 2023). Do ponto de vista histórico, essa área se insere em uma tradição mais ampla de composição algorítmica e de sistemas de geração musical automática, que vai desde abordagens baseadas em regras e modelos probabilísticos até métodos de otimização e composição assistida por computador (NIERHAUS, 2009; MIRANDA, 2001). Mais recentemente, levantamentos abrangentes propuseram taxonomias funcionais para classificar sistemas de música generativa de acordo com suas finalidades (por exemplo, composição, acompanhamento, harmonização, improvisação), evidenciando a diversidade de tarefas e estratégias empregadas (HERREMANS; CHUAN; CHEW, 2017). Em paralelo, pesquisas de revisão focadas em técnicas de aprendizado profundo e em ferramentas de IA para geração musical destacam tanto o crescimento do campo quanto a variedade de representações musicais, arquiteturas de modelos e aplicações práticas envolvidas (JI; YANG; LUO, 2023; ZHU et al., 2023). Nesse contexto, técnicas de otimização combinatória, especialmente aquelas baseadas em programação matemática, têm se mostrado promissoras para capturar regras estruturais e estilísticas subjacentes à composição musical (TANAKA; FUJII, 2015). Dentre as vertentes da chamada música generativa, destaca-se o desafio de gerar trechos musicalmente coerentes, como solos de guitarra, a partir de critérios objetivos que refletem preferências estilísticas humanas.

Inspirado por esse cenário interdisciplinar, o presente trabalho explora a formulação da tarefa de geração de solos de guitarra como um problema de otimização em grafos, no qual os elementos musicais básicos — os *licks* — são modelados como nós conectados por transições ponderadas que refletem sua adequação musical. A qualidade das transições é quantificada com base em avaliações subjetivas de músicos profissionais e amadores, previamente coletadas no estudo de Cunha, Subramanian e Herremans (2018), permitindo que o modelo busque sequências de *licks* que maximizem a qualidade musical do solo.

Os *licks* podem ser compreendidos como uma combinação de fragmentos melódicos, formando "frases musicais". São grupos de notas musicais que podem ser utilizados em diferentes solos de acordo com a harmonia a ser tocada. Neste trabalho, todos os *licks* provenientes do *dataset* utilizados estão no tom de Dó, de modo que os solos resultantes sempre atendam a esse tom. Mais conceitos de teoria musical relevantes para este trabalho estão presentes na Seção 2.1.

Para formalizar esse problema, são utilizadas ferramentas de programação linear inteira-mista, com suporte da biblioteca PuLP em Python e resolução via *solvers* como CBC, que implementam o método *branch-and-cut*. A modelagem matemática do problema descreve, essencialmente, uma função objetivo que se deseja minimizar (custos de transição entre *licks*) e restrições relacionadas à quantidade de vezes que cada *lick* pode ser utilizado, a duração em compassos do solo, quantidades máximas de *licks* de determinadas categorias, quais categorias

de *licks* que o primeiro e último *licks* devem pertencer e a impossibilidade de existir subciclos no grafo composto por *licks* que representa a solução. Mais detalhes dessa modelagem estão na Seção 4.2.2.

Além de reproduzir os experimentos de Cunha, Subramanian e Herremans (2018), o trabalho explora em detalhes a modelagem matemática envolvida, analisa a eficiência dos algoritmos utilizados e discute alternativas heurísticas para possíveis extensões futuras. Os solos gerados são, ao final, exportados em formato compatível com editores musicais como o *Guitar Pro*, possibilitando sua avaliação prática.

Dessa forma, este trabalho não apenas contribui para o desenvolvimento de ferramentas computacionais voltadas à composição musical, como também serve como estudo de caso didático sobre a aplicação de técnicas de otimização em um domínio não convencional, unindo música e Ciência da Computação.

1.1 OBJETIVOS

O presente trabalho tem como objetivo geral analisar e replicar o experimento do artigo Cunha, Subramanian e Herremans (2018). Para tal, podem ser definidos os seguintes objetivos específicos:

1. Estudo dos conceitos de otimização envolvidos;
2. Estudo detalhado do artigo de Cunha, Subramanian e Herremans (2018);
3. Implementação da solução de Cunha, Subramanian e Herremans (2018);
4. Comparação dos resultados obtidos neste trabalho com os obtidos por Cunha, Subramanian e Herremans (2018);
5. Análise de sensibilidade dos parâmetros do problema em relação ao custo computacional.

2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção são introduzidos alguns conceitos relevantes para o presente trabalho. O conteúdo consiste em descrever alguns dos componentes nos quais a modelagem do problema se baseia. Além disso, busca-se descrever como os algoritmos utilizados para a resolução do problema funcionam. São considerados tópicos de teoria musical, o problema clássico do caixeiro viajante (TSP – *Traveling Salesman Problem*), problema do caminho mais curto elementar com restrições de recursos (*Elementary Shortest Path Problem with Resource Constraints* - ESP-PRC) e três métodos de solução para problemas de programação inteira: *branch-and-bound*, planos de corte e *branch-and-cut*.

2.1 CONCEITOS DE TEORIA MUSICAL

Um solo de guitarra pode ser visto como uma combinação de pequenos fragmentos melódicos chamados *licks*. Esses *licks* são como “frases musicais prontas”, isto é, grupos de notas ou sequências que o guitarrista pode encaixar em diferentes solos, dependendo da harmonia que está sendo tocada (FISHER, 1995). Usar *licks* como blocos de construção tem uma vantagem importante: cada um deles já incorpora detalhes expressivos típicos da guitarra, como *bends* (quando a corda é esticada para mudar o tom), *slides* (quando o dedo desliza de uma nota para outra), *hammer-ons* e *pull-offs* (técnicas que facilitam a ligação entre notas). Na prática, esses *licks* podem ser vistos como unidades mínimas de vocabulário musical. Assim como um falante combina palavras e expressões prontas para formar frases, o guitarrista combina *licks* para construir um discurso musical coerente. Cada *lick* carrega não apenas informação melódica (quais notas são tocadas), mas também rítmica (em que momentos essas notas acontecem) e expressiva (como cada nota é articulada).

Em música tonal, as notas se organizam em torno de um centro chamado tonalidade ou tom (por exemplo, Dó maior ou Lá menor). Dentro de cada tonalidade, certas notas são mais estáveis (como a tônica) e outras funcionam como notas de passagem ou de tensão. Os *licks* usados em solos geralmente são construídos a partir de escalas associadas à tonalidade e aos acordes da harmonia. No contexto do *blues* e do *rock*, duas escalas são especialmente relevantes: a escala pentatônica menor e a escala de *blues*. A escala pentatônica menor utiliza cinco notas que evitam intervalos considerados mais dissonantes, o que facilita a improvisação sobre uma grande variedade de acordes. Já a escala de *blues* acrescenta a chamada *blue note* (um grau alterado da escala, normalmente a quinta diminuta ou a terça menor “abaixada”), produzindo a sonoridade característica do gênero. Muitos *licks* de *blues* são essencialmente padrões recorrentes nessas escalas; por isso, conhecer essas estruturas é fundamental para descrever e manipular *licks* de forma sistemática. Além disso, há uma relação direta entre acordes e escalas: sobre cada acorde da progressão, o guitarrista tende a escolher *licks* baseados em escalas compatíveis com aquele acorde. Por exemplo, sobre um acorde de Dó7 (C7) em um *blues* de Dó, é comum utilizar *licks* baseados na escala de Dó mixolídio ou na pentatônica de Dó, eventualmente com

a *blue note*.

Outro elemento central para entender *licks* é o ritmo. A maior parte dos *blues* e do *rock* é escrita em compasso 4/4, ou seja, cada compasso é dividido em quatro tempos. Dentro desses tempos, as notas podem ser subdivididas de forma binária (em colcheias “retas”) ou ternária (em colcheias com *swing* ou *shuffle*), o que altera profundamente a sensação rítmica do solo. Os *licks* não são apenas sequências de alturas, mas padrões rítmicos específicos: a posição exata das notas dentro do compasso (início do tempo, contratempo, síncopas) contribui para a sensação de antecipação, repouso ou tensão. Em gêneros como o *blues*, é comum acentuar os tempos 2 e 4 (os chamados *backbeats*), o que também influencia como os *licks* são construídos e percebidos. Para esta pesquisa, esse aspecto rítmico é relevante porque a escolha e a organização dos *licks* ao longo dos compassos devem respeitar tanto a divisão métrica quanto a acentuação típica do estilo. Assim, um *lick* pode ser mais apropriado para iniciar um compasso, enquanto outro funciona melhor como fechamento de frase ou “resposta”.

Neste trabalho, adotamos os *licks* como unidades fundamentais para compor solos de guitarra completos e coerentes. Essa abordagem facilita, por exemplo, a aplicação de restrições ligadas à estrutura do solo — como no caso do *blues* de 12 compassos, uma das progressões de acordes mais tradicionais desse estilo. O *blues* de 12 compassos pode ser imaginado como um “ciclo” de 12 medidas que se repete. Ele é construído em cima de três acordes principais de uma tonalidade: o I, o IV e o V (no tom de Dó, seriam os acordes de Dó, Fá e Sol). A sequência típica segue este padrão: I – I – I – I – IV – IV – I – I – V – IV – I – V (KRENZ, 2010). Além disso, é comum substituir acordes maiores por acordes de sétima da dominante, que dão ao *blues* sua sonoridade característica. Esse tipo de acorde acrescenta uma “nota de tensão” (a sétima menor) que pede naturalmente por resolução, criando o clima inconfundível do gênero. Em muitos casos, a forma do *blues* de 12 compassos também é organizada em frases do tipo “pergunta–pergunta–resposta” (estrutura AAB), em que os primeiros oito compassos apresentam uma ideia que é repetida ou levemente variada, e os quatro últimos compassos trazem uma conclusão ou *turnaround*.

Todos esses conceitos — *licks*, tonalidade, escalas, ritmo, compasso e forma — são usados neste trabalho de maneira operacional. Cada *lick* da base de dados pode ser associado a informações como tonalidade, acorde sobre o qual ele foi originalmente tocado, posição dentro do compasso e função dentro da forma (por exemplo, início de frase, desenvolvimento ou *turnaround*). Dessa forma, é possível selecionar e combinar *licks* respeitando tanto a harmonia (a sequência de acordes do *blues* de 12 compassos) quanto a estrutura rítmica e formal típica do estilo, produzindo solos que, ainda que gerados automaticamente, mantêm coerência musical com a prática do guitarrista humano.

2.2 PROBLEMA DO CAIXEIRO VIAJANTE (TSP)

O problema do caixeiro viajante (APPLEGATE et al., 2011) é um problema especial em Ciência da Computação. A definição do problema é simples. O caixeiro viajante deve visitar

cada cidade em seu território exatamente uma vez e depois retornar ao seu ponto de partida. Dado o custo da viagem entre cada par de cidades, a tarefa é organizar um passeio completo que atinja o custo total mínimo. Um possível espaço de busca para o TSP é um conjunto de permutações de n cidades. Qualquer permutação única de n cidades produz uma solução, que é interpretada como um passeio completo, com um retorno para casa implícito assim que a última cidade da lista for visitada. O tamanho deste espaço de busca é $O(n!)$.

Considera-se um grafo dirigido completo $G = (V, A)$, com $V = \{i \in \mathbb{N} \mid 1 \leq i \leq n\}$ e custos $c_{ij} \geq 0$ para cada arco $(i, j) \in A$, com $i \neq j$. Definem-se variáveis binárias x_{ij} que valem 1 se o arco (i, j) é escolhido no passeio e 0 caso contrário.

Uma possível formulação do problema é apresentada em (2.1) - (2.5).

$$\min \sum_{i \in V} \sum_{\substack{j \in V \\ j \neq i}} c_{ij} x_{ij} \quad (2.1)$$

$$\text{s.a.} \quad \sum_{\substack{j \in V \\ j \neq i}} x_{ij} = 1 \quad \forall i \in V \quad (2.2)$$

$$\sum_{\substack{i \in V \\ i \neq j}} x_{ij} = 1 \quad \forall j \in V \quad (2.3)$$

$$\sum_{i \in S} \sum_{\substack{j \in S \\ j \neq i}} x_{ij} \leq |S| - 1 \quad \forall S \subset V, 2 \leq |S| \leq n - 2 \quad (2.4)$$

$$x_{ij} \in \{0, 1\} \quad \forall (i, j) \in A. \quad (2.5)$$

As restrições (2.2) e (2.3) garantem que cada cidade tenha exatamente uma saída e uma entrada. As desigualdades (2.4) eliminam subciclos próprios (ciclos que não visitam todas as cidades).

No problema principal deste trabalho, em vez de cidades, são considerados *licks*, o custo de viagem de uma cidade para outra é o custo de transição de um *lick* para outro e, em ambos, para um determinado número de nodos, busca-se o caminho de custo mínimo. Diferentemente do TSP, no qual todas as cidades do conjunto analisado devem ser visitadas exatamente uma vez, no problema deste trabalho, dentre os *licks* do conjunto analisado, um número menor ou igual ao total de *licks* (de preferência, um valor que gere 12 compassos, acordando com a Seção 2.1) deve ser escolhido e esses *licks* provenientes desse subconjunto devem ser visitados exatamente uma vez. Além disso, há mais restrições no problema deste trabalho com respeito à escolha dos nodos que irão compor o caminho de custo mínimo.

2.3 PROBLEMA DO CAMINHO MAIS CURTO ELEMENTAR COM RESTRIÇÕES DE RECURSOS (ESPPRC)

O problema do caminho mais curto com restrições de recursos (SPPRC, do inglês *Shortest Path Problem with Resource Constraints*) consiste em encontrar, em um grafo dirigido $G = (V, A)$, um caminho de custo mínimo entre um nó de origem s e um nó de destino t , sujeito a

limites sobre um ou mais recursos (por exemplo, tempo, capacidade, energia, número de visitas, etc.). Cada arco $(i, j) \in A$ consome certa quantidade de recursos, e um caminho é considerado viável se, ao longo de sua trajetória, os recursos acumulados permanecem dentro de intervalos previamente definidos.

Na versão clássica do SPPRC, admite-se que o caminho possa visitar um mesmo nó mais de uma vez, desde que as restrições de recursos não sejam violadas. O *problema do caminho mais curto elementar com restrições de recursos* (ESPPRC ou ESPRC) impõe, adicionalmente, a *elementaridade* do caminho: cada vértice pode ser visitado, no máximo, uma vez. Em outras palavras, o caminho procurado é simples (livre de ciclos), além de respeitar os limites de recursos. Essa exigência reflete de forma mais fiel diversos problemas de roteirização, em que não faz sentido visitar o mesmo cliente ou tarefa mais de uma vez, mas aumenta significativamente a complexidade computacional do problema (FEILLET et al., 2004).

O ESPPRC surge com frequência como subproblema em abordagens de decomposição (por exemplo, *branch-and-price*) para problemas de roteirização de veículos, alocação de tarefas e programação de serviços com janelas de tempo. Em tais contextos, cada solução viável do ESPPRC corresponde a uma rota elementar factível em termos de recursos, e o custo dessa rota é utilizado na geração de colunas com custo reduzido negativo.

Consideram-se custos c_{ij} , tempos de deslocamento $d_{ij} \geq 0$, tempos de serviço $w_i \geq 0$ e janelas de tempo $[e_i, \ell_i]$ para $i \in V$. Seja M suficientemente grande, $x_{ij} \in \{0, 1\}$ indica uso do arco (i, j) e $t_i \geq 0$ é o tempo de chegada ao nó i partindo-se da origem s . Um modelo de programação inteira mista para o ESPPRC com janelas de tempo pode ser escrito como em (2.6)–(2.19):

$$\min \sum_{(i,j) \in A} c_{ij} x_{ij} \quad (2.6)$$

$$\text{s.a.} \quad \sum_{j:(s,j) \in A} x_{sj} = 1 \quad (2.7)$$

$$\sum_{i:(i,s) \in A} x_{is} = 0 \quad (2.8)$$

$$\sum_{i:(i,t) \in A} x_{it} = 1 \quad (2.9)$$

$$\sum_{j:(t,j) \in A} x_{tj} = 0 \quad (2.10)$$

$$\sum_{j:(i,j) \in A} x_{ij} - \sum_{j:(j,i) \in A} x_{ji} = 0 \quad \forall i \in V \setminus \{s, t\} \quad (2.11)$$

$$\sum_{j:(i,j) \in A} x_{ij} \leq 1 \quad \forall i \in V \setminus \{s, t\} \quad (2.12)$$

$$\sum_{j:(j,i) \in A} x_{ji} \leq 1 \quad \forall i \in V \setminus \{s, t\} \quad (2.13)$$

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1 \quad \forall S \subset V \setminus \{s, t\}, 2 \leq |S| \leq n - 2 \quad (2.14)$$

$$t_s = e_s \quad (2.15)$$

$$t_i \geq e_i \sum_{j:(j,i) \in A} x_{ji} \quad \forall i \in V \setminus \{s\} \quad (2.16)$$

$$t_i \leq \ell_i \sum_{j:(j,i) \in A} x_{ji} \quad \forall i \in V \setminus \{s\} \quad (2.17)$$

$$t_j \geq t_i + d_{ij} + w_j - M(1 - x_{ij}) \quad \forall (i, j) \in A \quad (2.18)$$

$$x_{ij} \in \{0, 1\}, \quad t_i \geq 0 \quad \forall (i, j) \in A, i \in V. \quad (2.19)$$

A função objetivo (2.6) minimiza o custo total do caminho de s até t . As restrições (2.7)–(2.10) forçam exatamente uma saída de s e uma chegada a t , impedindo arcos de entrada em s e de saída de t . As equações de balanço de fluxo em (2.11) garantem a conservação de fluxo nos vértices intermediários, isto é, sempre que um nó é visitado, exatamente um arco entra e um arco sai desse nó.

As restrições de grau (2.12)–(2.13) limitam o número de arcos incidentes a cada nó (exceto s e t) a, no máximo, um de entrada e um de saída, o que impede múltiplas visitas ao mesmo vértice. Em conjunto com o balanço de fluxo, essas restrições já reduzem fortemente a ocorrência de ciclos. No entanto, ainda podem existir subciclos desconectados do caminho entre s e t . Por isso, introduzem-se as desigualdades de eliminação de subciclos (2.14), do tipo Dantzig–Fulkerson–Johnson, que asseguram que nenhum subconjunto próprio $S \subset V \setminus \{s, t\}$ forma um circuito fechado de arcos. Essas restrições são responsáveis por garantir a *elementaridade* do caminho, mas são em número exponencial e, na prática, costumam ser tratadas via planos de corte ou geração dinâmica.

As restrições (2.15)–(2.17) modelam as janelas de tempo. A igualdade (2.15) fixa o tempo de partida da origem em e_s . As desigualdades (2.16)–(2.17) asseguram que, se o nó i é visitado (isto é, se algum arco entra em i), então o tempo de chegada t_i pertence ao intervalo $[e_i, \ell_i]$. Note que o somatório dos arcos de entrada funciona como um “indicador” de visita ao nó: se o nó não é visitado, essas restrições ficam automaticamente inativas.

A propagação do recurso “tempo” ao longo do caminho é feita por (2.18). Para qualquer arco (i, j) , se $x_{ij} = 1$, a desigualdade obriga que o tempo de chegada a j seja, pelo menos, o tempo de chegada a i acrescido do tempo de deslocamento d_{ij} e do tempo de serviço w_j . Caso o arco não seja utilizado ($x_{ij} = 0$), o termo com M torna a restrição redundante. Trata-se de uma formulação padrão com constante “grande o suficiente” (M), que lineariza a relação entre as variáveis de decisão binárias e os tempos de chegada. Por fim, (2.19) define os domínios das variáveis.

Embora o modelo acima esteja escrito apenas para o recurso tempo, a mesma lógica pode ser estendida para múltiplos recursos (capacidade, nível de bateria, número máximo de visitas, etc.), introduzindo-se variáveis análogas a t_i e restrições de propagação apropriadas. Nesse sentido, o ESPPRC fornece um arcabouço geral para problemas em que se busca um caminho simples de custo mínimo em um grafo, respeitando simultaneamente limites de recursos ao longo da rota.

2.4 MÉTODOS DE SOLUÇÃO

Existem diversos métodos de solução para o problema de otimização proposto, podendo ser classificados como exatos e heurísticos. Os métodos heurísticos sugerem soluções aproximadas para o problema, porém apresentam, em muitos casos, um menor tempo de execução. Dentre os métodos heurísticos podemos citar busca tabu, busca local iterada, algoritmo genético e *simulated annealing*. Entretanto, neste trabalho o foco está nos métodos exatos, sendo estes utilizados para a resolução do problema em questão. Assim, adiante serão apresentados alguns desses métodos. O leitor interessado em métodos heurísticos pode consultar Gendreau, Potvin et al. (2010).

2.4.1 Preliminares de um problema de otimização

Antes da introdução do primeiro método exato de otimização, será feita uma breve introdução aos conceitos básicos de um problema de otimização. Para isso, considera-se o problema P como exemplo (ARENALES et al., 2007).

$$(P) \quad z = \max(5x_1 - x_2) \quad (2.20)$$

$$7x_1 - 5x_2 \leq 13 \quad (2.21)$$

$$3x_1 + 2x_2 \leq 17 \quad (2.22)$$

$$x \in \mathbb{Z}_+^2 \quad (2.23)$$

Neste exemplo, em (2.20) existe uma função objetivo z que se deseja maximizar. Em (2.21) - (2.22), as desigualdades representam as restrições que as variáveis presentes na função objetivo devem obedecer para que uma solução seja considerada válida. Em (2.23), tem-se a declaração do domínio que as variáveis podem assumir para compor uma solução válida. Neste caso, x_1 e x_2 pertencem ao conjunto dos inteiros não-negativos.

Portanto, busca-se a escolha de valores inteiros não-negativos para x_1 e x_2 de modo a maximizar a função objetivo z , atendendo-se as restrições $7x_1 - 5x_2 \leq 13$ e $3x_1 + 2x_2 \leq 17$.

2.4.2 *Branch-and-bound*

Este método é utilizado para a resolução de problemas de programação inteira, que são problemas de otimização cujas variáveis presentes na função objetivo podem assumir apenas valores inteiros para que a solução seja válida. Primeiramente, aplica-se uma relaxação linear ao problema original, procedimento que modifica o domínio das variáveis, de modo que estas que até então estavam no domínio dos números inteiros, passam para o domínio dos números racionais, ou seja, soluções com variáveis com valores fracionários são aceitas e o problema de programação inteira é transformado em um problema de programação linear. Em seguida, resolve-se o problema de programação linear por meio de determinado método, como o Simplex (ARENALES et al., 2007). Quando resolvido, o valor da função objetivo encontrado passa a ser o limitante superior (em problemas de maximização) para as possíveis futuras soluções, isto é, caso o método seja continuado, todas as soluções encontradas não gerarão uma função objetivo com valor superior ao valor recém encontrado.

Se a solução for infactível, esta é descartada. Se a solução for factível e não for inteira, escolhe-se uma variável fracionária da solução encontrada. Além disso, caso o valor encontrado seja melhor que o valor encontrado para uma solução inteira, este valor é adotado como limitante superior ou inferior para a solução final para funções de maximização ou minimização, respectivamente. Então, criam-se dois subproblemas contendo restrições adicionais de piso/teto (para a variável fracionária escolhida) ao problema recém resolvido.

Se a solução for factível e inteira, esta é comparada com as soluções inteiras já encontradas. Se a solução for melhor que as anteriores, é tida como melhor solução atual e não serão criados mais subproblemas a partir do problema recém resolvido. Caso contrário, simplesmente não serão criados mais subproblemas a partir do problema recém resolvido. O algoritmo que corresponde a esse procedimento é mostrado no Algoritmo 1.

Algoritmo 1 – branch-and-bound

Entrada: problema original de programação inteira P (nó 0 da árvore B&B).

Saída: solução ótima x^* (se existir) e valor z^* .

Passo 0 (Inicialização). Faça $\bar{z} = \infty$, $z^* = -\infty$, $x^* = \emptyset$, $L = \{P\}$.

Passo 1 (Seleção de nó). Selecione um nó *ativo* i , associado ao subproblema P^i , da lista L . Se L estiver vazia, vá para o **Passo 6**.

Passo 2 (Teste de eliminação 1). Se a região factível de PL^i (relaxação linear de P^i) for vazia, vá para o **Passo 1**.

Passo 3 (Teste de eliminação 2). Resolva PL^i e obtenha o valor ótimo $\bar{z}^i$. Se $\bar{z}^i \leq z^*$, vá para o **Passo 1**.

Passo 4 (Teste de eliminação 3). Se a solução ótima $\bar{x}_i$ de PL^i é inteira com valor $\bar{z}^i$, então

1. se $\bar{z}^i > z^*$, atualize $x^* \leftarrow \bar{x}_i$ e $z^* \leftarrow \bar{z}^i$;
2. elimine da lista L todos os nós ativos j tais que $\bar{z}^j \leq z^*$;

e vá para o **Passo 1**.

Passo 5 (Ramificação). Escolha uma variável de $\bar{x}_i$ que seja fracionária. Crie dois subproblemas a partir de P^i , adicionando as restrições $x_k \leq \lfloor \bar{x}_{i,k} \rfloor$ e $x_k \geq \lceil \bar{x}_{i,k} \rceil$, respectivamente. Adicione os dois novos problemas à lista L e vá para o **Passo 1**.

Passo 6 (Fim). Se $z^* = -\infty$, então não existe solução factível; caso contrário, a solução incumbente x^* é ótima.

O processo é repetido recursivamente para cada subproblema até que não se possa mais criar subproblemas a partir do problema atual.

Como exemplo, considera-se o novamente o problema de programação inteira P Arenales et al. (2007) aplica-se o método *branch-and-bound* para resolvê-lo.

$$(P) \quad z = \max (5x_1 - x_2) \quad (2.20)$$

$$7x_1 - 5x_2 \leq 13 \quad (2.21)$$

$$3x_1 + 2x_2 \leq 17 \quad (2.22)$$

$$x \in \mathbb{Z}_+^2 \quad (2.23)$$

Relaxando-se linearmente o problema P , obtém-se o gráfico mostrado na Figura 1.

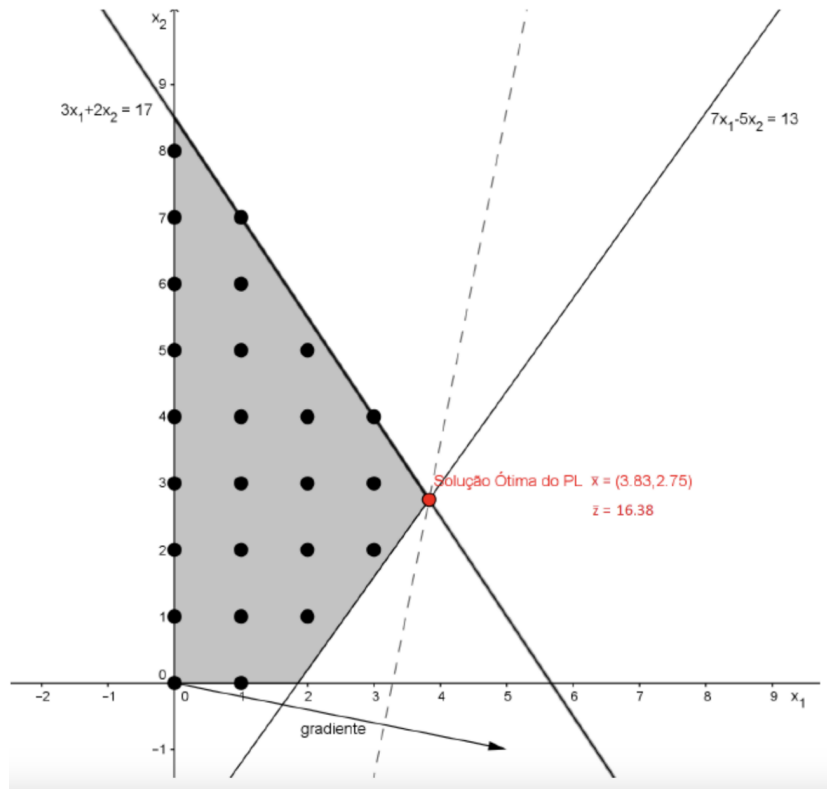


Figura 1 – Obtenção de uma solução ótima $\bar{z} = 16,38$ para o problema relaxado PL para $\bar{x}_1 = 3,83$ e $\bar{x}_2 = 2,75$. O vetor gradiente $\nabla z = (5, -1)$ aponta para a direção em que a função objetivo z cresce mais rapidamente. (Arenales et al., 2007)

As restrições $7x_1 - 5x_2 \leq 13$, $3x_1 + 2x_2 \leq 17$ e $x \in \mathbb{Z}_+^2$ delimitam a região onde existem soluções factíveis e escolhe-se, dentre os vértices do polígono que representa a região de soluções factíveis, aquele cujo valor da função objetivo $\bar{z}$ de PL seja o maior. Desse modo, tem-se como limitante superior 16,38 até o momento. Em seguida, opta-se pela escolha da variável x_1 para a geração de novos dois subproblemas PI_1 , com a nova restrição $x_1 \leq 3$, e PI_2 , com a nova restrição $x_1 \geq 4$ (ARENALES et al., 2007).

PI_1	PI_2
$z_1 = \max \quad 5x_1 - x_2$	$z_2 = \max \quad 5x_1 - x_2$
s.a. $7x_1 - 5x_2 \leq 13$	s.a. $7x_1 - 5x_2 \leq 13$
$3x_1 + 2x_2 \leq 17$	$3x_1 + 2x_2 \leq 17$
$x_1 \leq 3$	$x_1 \geq 4$
$x_1 \in \mathbb{Z}_+$	$x_1 \in \mathbb{Z}_+$
$x_2 \in \mathbb{Z}_+$	$x_2 \in \mathbb{Z}_+$

O processo descrito até então pode ser representado como uma árvore binária, conforme a Figura 2.

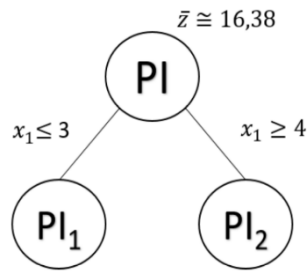


Figura 2 – Problema inicial de programação inteira PI (ou P) é relaxado, resolvido e dividido em dois novos subproblemas de programação inteira PI_1 e PI_2 , cada um com novas restrições. (Arenales et al., 2007)

Assim, aplica-se uma relaxação linear em PI_1 , resultando em PL_1 e obtendo-se a uma nova solução $\bar{x}_1 = 3$ e $\bar{x}_2 = 1,59$ para uma função objetivo $\bar{z} = 13,07$. Ao se iniciar o procedimento de resolução de PI_2 , percebe-se que a nova restrição $x_1 \geq 4$ delimita uma região que não faz parte da região de soluções factíveis, tornando assim, PI_2 infactível e impossível de ser dividido em dois novos subproblemas. Então, 13,07 é tido como o novo limitante superior. O gráfico da Figura 3 evidencia esse processo.

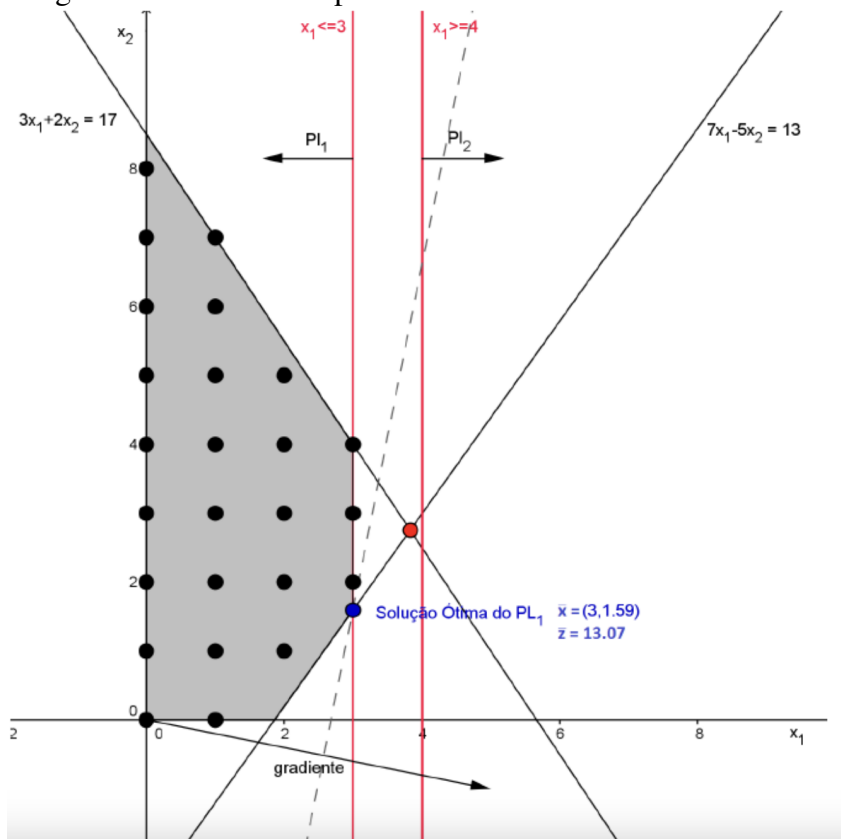


Figura 3 – PI_1 é relaxado linearmente, tornando-se PL_1 , que é resolvido, obtendo-se $\bar{x}_1 = 3$ e $\bar{x}_2 = 1,59$, com a função objetivo $\bar{z} = 13,07$. PI_2 é infactível. (Arenales et al., 2007)

Então, ramifica-se o nodo PI_1 ou P_1 a partir da escolha da única variável fracionária

x_2 . Assim, cria-se dois subproblemas P_3 com a nova restrição $x_2 \leq 1$ e P_4 com a nova restrição $x_2 \geq 2$. No momento, a árvore encontra-se conforme a Figura 4.

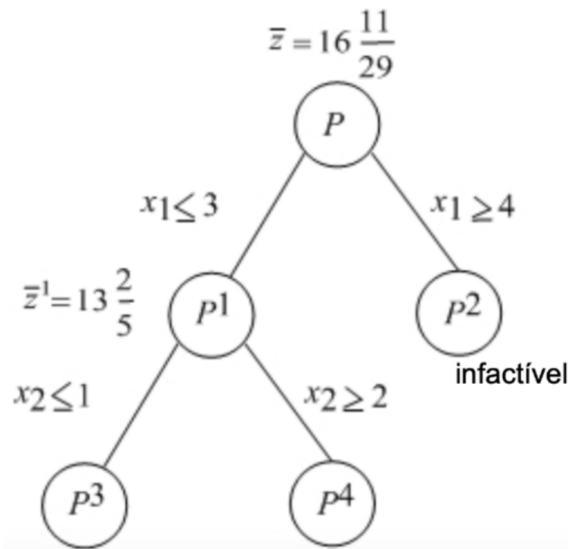


Figura 4 – P_1 gera dois novos subproblemas P_3 e P_4 . (Arenales et al., 2007)

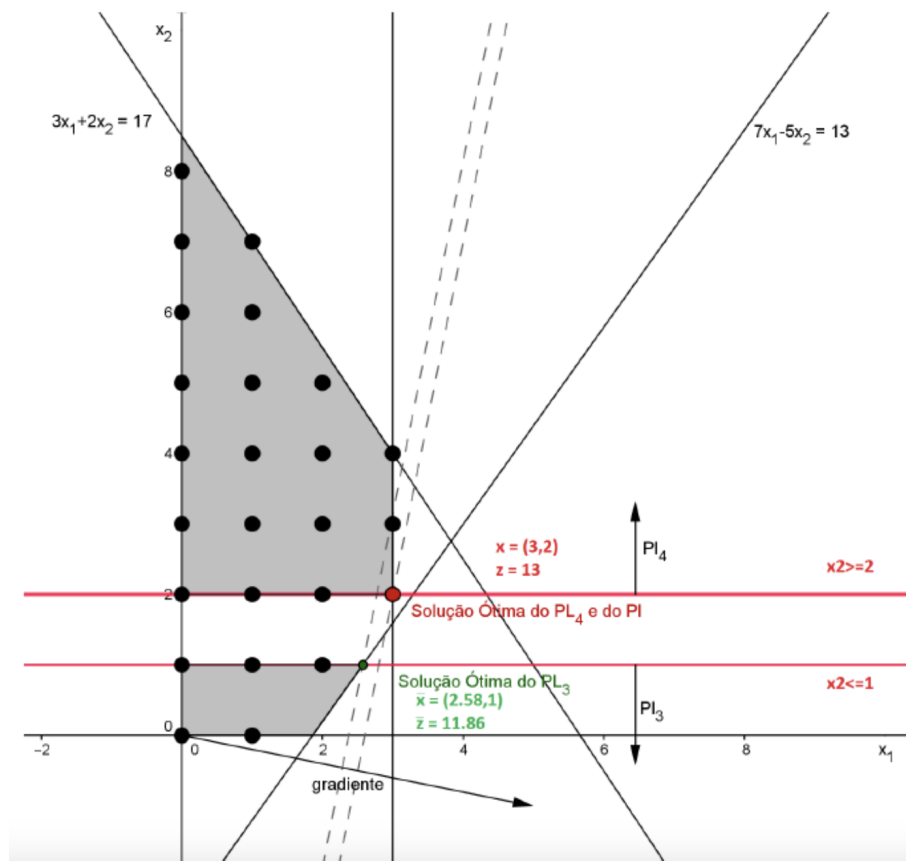


Figura 5 – P_3 e P_4 são relaxados linearmente em PL_3 e PL_4 , respectivamente, e a solução de PL_4 é adotada como solução para P . (Arenales et al., 2007)

Assim, aplica-se uma relaxação linear em P_3 , resultando em PL_3 e obtendo-se a uma nova solução $\bar{x}_1 = 2,58$ e $\bar{x}_2 = 1$ para uma função objetivo $\bar{z} = 11,86$. Do mesmo modo, aplica-se uma relaxação linear em P_4 , resultando em PL_4 e obtendo-se uma nova solução $\bar{x}_1 = 3$

e $\bar{x}_2 = 2$ para uma função objetivo $\bar{z} = 13$. Como todas as variáveis da solução de PL_4 são inteiras e a função objetivo de PL_3 é menor que a função objetivo de PL_4 , adota-se a solução de PL_4 como solução para o problema inicial P , que terá $x_1 = 3$, $x_2 = 2$ e função objetivo $z = 13$. O gráfico da Figura 5 mostra esse processo.

2.4.3 Planos de corte

Apesar da eficácia do método *branch-and-bound* na resolução de problemas de programação inteira, sua eficiência pode ser significativamente aprimorada quando combinada com técnicas que fortalecem a formulação do problema original. Nesse sentido, o método de planos de corte surge como uma abordagem complementar, que visa refinar a relaxação linear do problema adicionando desigualdades válidas, chamadas de cortes, que eliminam regiões da solução contínua que não contêm soluções inteiras, sem excluir nenhuma solução viável do problema original. Essa técnica baseia-se na identificação de restrições adicionais que estreitam o espaço viável do problema relaxado, tornando-o mais próximo da formulação inteira.

Existem procedimentos para a geração de cortes (Wolsey e Nemhauser (1999)) e, dentre eles, há o de Chvátal-Gomory, que será explorado no exemplo 1 na sequência.

Exemplo 1: Problema contendo duas variáveis e três restrições. (Wolsey, Nemhauser, 1988)

$$S = \{x \in \mathbb{Z}_+^n : Ax \leq b\}.$$

$$A = \begin{pmatrix} -1 & 2 \\ 5 & 1 \\ -2 & -2 \end{pmatrix}, \quad b = \begin{pmatrix} 4 \\ 20 \\ -7 \end{pmatrix}.$$

Segue-se o procedimento a seguir de modo a adicionar cortes à região factível, por meio da adição de novas restrições. A cada iteração, é gerado um novo corte de modo que o processo seja interrompido quando as soluções para o problema com a atual região factível sejam valores inteiros.

Para $S = \{x \in \mathbb{Z}_+^n : Ax \leq b\}$, onde $A = (a_1, a_2, \dots, a_n)$ e $N = \{1, \dots, n\}$, o procedimento contém três etapas:

- i. $\sum_{j \in N} (ua_j)x_j \leq ub, \quad u \geq 0;$
- ii. $\sum_{j \in N} (\lfloor ua_j \rfloor)x_j \leq ub, \quad x \geq 0$ implica $-\sum_{j \in N} (ua_j - \lfloor ua_j \rfloor)x_j \leq 0;$
- iii. $\sum_{j \in N} (\lfloor ua_j \rfloor)x_j \leq \lfloor ub \rfloor, \quad x \in \mathbb{Z}^n$ implica que $\sum_{j \in N} (\lfloor ua_j \rfloor)x_j$ é um inteiro.

O primeiro passo corresponde à escolha de um valor para u . Esta escolha é feita com base no método Simplex a cada iteração para se gerar uma nova restrição ao problema, ou seja, um novo corte à região factível. Em seguida, com base neste valor de u , manipula-se as

restrições existentes até então no problema, conforme o passo *i* e obtém-se uma nova restrição redundante para o problema.

Então, de acordo com o passo *ii*, obtém-se apenas a parte inteira dos coeficientes do lado esquerdo da equação. Por fim, no passo *iii*, obtém-se apenas a parte inteira dos valores do lado direito da equação.

Uma iteração do processo é exemplificada a seguir (Wolsey e Nemhauser (1999)).

$u = (\frac{5}{11}, \frac{3}{22}, 0)$ na etapa *i* resulta em:

$$\frac{5}{22}x_1 + \frac{23}{22}x_2 \leq \frac{100}{22}.$$

Então,

$$x_2 \leq \frac{100}{22} \quad (\text{etapa } ii)$$

e

$$x_2 \leq 4 \quad (\text{etapa } iii).$$

Deste modo, é gerada uma nova restrição para o problema original, resolve-se o novo problema tratando-o como um problema de programação linear, por meio de uma relaxação e encontra-se novas soluções. Se os valores das soluções forem inteiros, o método é interrompido e adota-se as soluções encontradas. Caso contrário, repete-se o processo com o uso de um novo valor para u .

2.4.4 *Branch-and-cut*

O método *branch-and-cut* é uma junção dos métodos *branch-and-bound* e planos de corte para a resolução problemas de programação inteira ou inteira-mista. Uma das vantagens oferecidas em relação ao *branch-and-bound*, é a construção de uma árvore com uma menor quantidade de nodos a fim de se encontrar ao menos uma solução ótima.

Inicialmente, o problema original é mapeado para o nodo raiz da árvore, sendo geradas soluções intermediárias iterativamente k vezes utilizando-se o método de planos de corte. Em seguida, caso não tenham sido encontradas soluções inteiras e factíveis, um novo limitante (superior ou inferior, para o problema de maximização ou minimização, respectivamente) é adotado e ocorre a ramificação da árvore em dois nodos. Em cada nodo, o método de planos de corte é utilizado e verifica-se se há soluções inteiras e factíveis. Em caso positivo, o método é interrompido, obtendo-se soluções inteiras para o problema original. Caso contrário, novas ramificações são feitas seguindo o *branch-and-bound*.

Vejamos um exemplo comparativo entre as árvores formadas pelos métodos *branch-and-bound* e *branch-and-cut*, no qual, no método *branch-and-cut*, o método de planos de corte é aplicado apenas no nodo raiz.

Considerando-se o problema de programação inteira do exemplo 2.

Exemplo 2 (Arenales et al. (2007))

$$z = \max 25x_1 + 2x_2 + 31x_3 + 30x_4$$

$$14x_1 + 25x_2 + 14x_3 + 6x_4 \leq 146$$

$$19x_1 + 30x_2 + 24x_3 + 29x_4 \leq 239$$

$$x \in \mathbb{Z}_+^4$$

Com o uso do método *branch-and-bound*, obtém-se a árvore da Figura 6, cujas duas soluções ótimas são

$$x_1 = 1, x_2 = 0, x_3 = 9, x_4 = 0,$$

$$x_1 = 6, x_2 = 0, x_3 = 4, x_4 = 1,$$

com valor $z = 304$

Utilizando-se o método *branch-and-cut* apenas no nó raiz, o limitante superior passa de 309,7 para 308,8 e obtém-se as duas soluções ótimas após a criação de apenas mais quatro nodos na árvore, como mostrado na Figura 7.

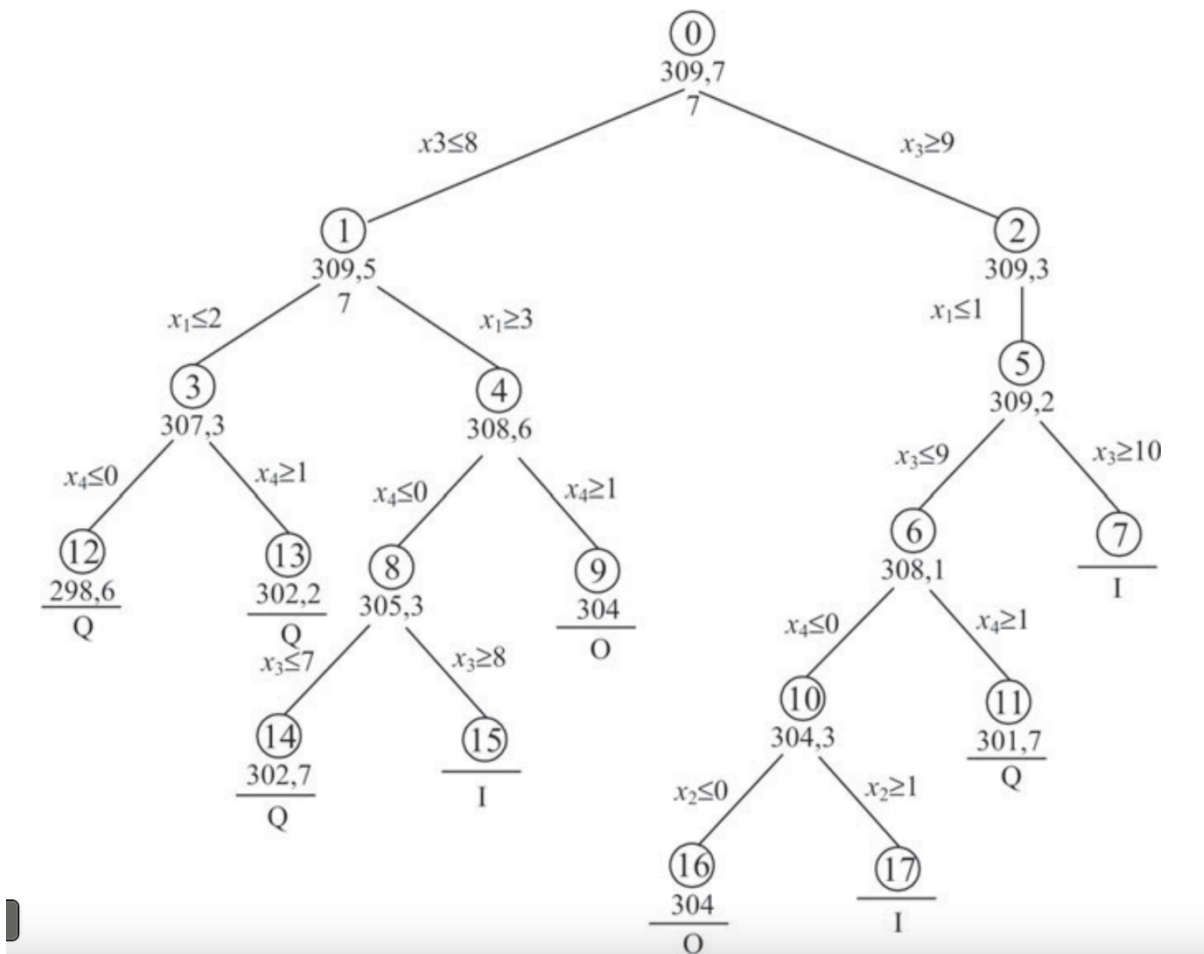


Figura 6 – Árvore de soluções para o exemplo 2 gerada a partir do método *branch-and-bound*. (Arenales et al., 2007)

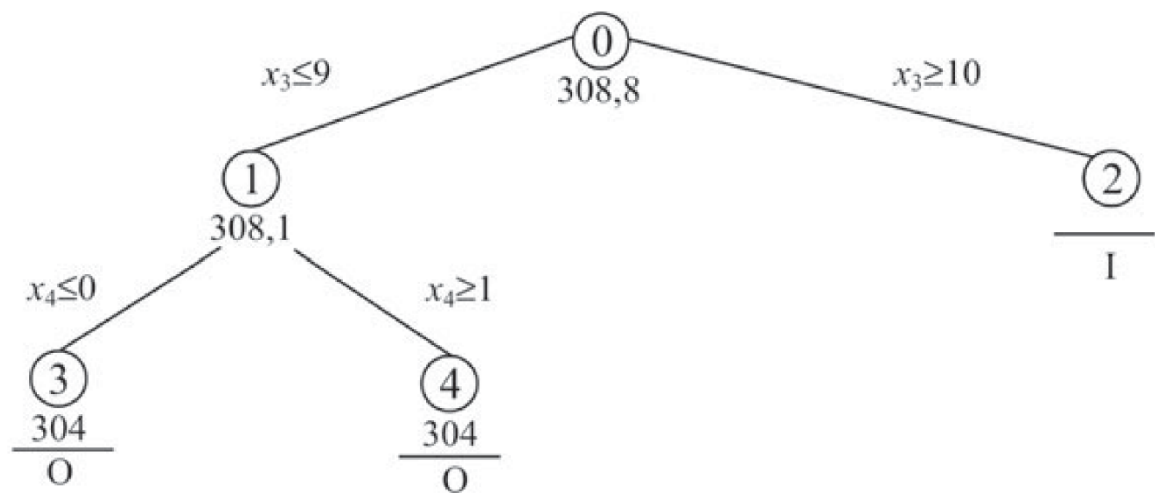


Figura 7 – Aplicação do método *branch-and-cut* para a resolução do exemplo 2, evidenciando uma menor quantidade de nodos na árvore de solução quando comparado ao método *branch-and-bound* para o mesmo problema. (Arenales et al., 2007)

Vale ressaltar que existem variações do método *branch-and-cut*, cada uma adotando condutas específicas, como a escolha do valor de k e do caminho a se seguir após uma nova ramificação na árvore.

3 TRABALHOS RELACIONADOS

Observa-se diferentes abordagens para a produção de música generativa ao longo dos anos, incluindo trabalhos como os de Cope (1996), que desenvolveu um programa que recebe dados relacionados aos trabalhos de um determinado compositor e quebra-os em fragmentos pequenos constituídos de intervalos e padrões harmônicos. Depois, padrões recorrentes e recursos estilísticos são identificados, gerando-se a gramática composicional do compositor, ou seja, o conjunto de regras para a produção musical que o compositor possivelmente adotaria, seguindo-se para a fase de recombinação dos fragmentos baseando-se nas regras de produção musical. Cope utilizou técnicas de inteligência artificial simbólica, adotando regras explícitas e representações musicais simbólicas para a confecção de seus experimentos.

Pachet (2003) produziu o Continuator, um sistema interativo de geração de música que foi projetado para preencher a lacuna entre duas abordagens tradicionais em tecnologia musical: sistemas interativos, que reagem em tempo real, mas carecem de profundidade estilística; e sistemas de imitação, que podem imitar estilos musicais, mas não são interativos. O objetivo do Continuator é permitir a interação musical em tempo real, mantendo-se a consistência estilística, permitindo que um músico "toque" com um computador em seu próprio estilo ou em outro estilo aprendido. Baseando-se em modelos de Markov de ordem variável que aprendem padrões musicais de um artista em tempo real, utilizando-se árvores de prefixos para armazenar subsequências de frases MIDI de entrada e suas possíveis continuações, o sistema segmenta a entrada do músico em frases e responde gerando continuações estilisticamente consistentes em tempo real, isto é, o sistema reage instantaneamente ao que é tocado.

Além do exemplo de Cunha, Subramanian e Herremans (2018) no qual o presente trabalho fortemente se baseia, demais abordagens de programação inteira e linear como a de Tanaka e Fujii (2015) consideram a música como sequências de padrões localizados, como acordes, padrões rítmicos e padrões melódicos. Em particular, gerar sequências controlando as relações entre padrões locais e estruturas globais é um problema difícil e em aberto. Enquanto abordagens gramaticais, que examinam estruturas globais, podem ser usadas para analisar como um fragmento é construído, elas não são necessariamente projetadas para gerar novos fragmentos controlando as características das estruturas globais, como a redundância de uma sequência ou a distribuição estatística de padrões específicos. Para lidar com esse problema, adotou-se uma abordagem baseada em programação inteira e foi mostrado que algumas características importantes das estruturas globais podem ser descritas apenas por igualdades e desigualdades lineares. Ainda no artigo de Tanaka e Fujii (2015), foi proposta uma formulação para a geração de sequências de padrões musicais controlando algumas estruturas globais das sequências, com foco especial na hierarquia e no grau de redundância em cada nível. Mostrou-se que tais estruturas podem ser expressas apenas por restrições lineares, necessárias para aplicar a programação linear. Trabalhos futuros incluem a descrição mais abrangente das estruturas globais, a implementação das restrições e a geração efetiva de novos fragmentos, a definição de restrições para as relações entre melodia e acordes e a definição das restrições para relações polifônicas.

Tabela 1 – Trabalhos relacionados em ordem cronológica

Ano	Objetivo / Contribuição Principal	Autor(es) / Sistema	Abordagem / Técnicas
1996	Criar músicas no estilo de um compositor a partir da recombinação de fragmentos extraídos de suas obras.	Cope	IA simbólica; regras explícitas; gramática composicional.
2003	Gerar música interativa em tempo real, preservando a consistência estilística do músico.	Pachet – Continuator	Modelos de Markov de ordem variável; árvores de prefixos.
2009	Criar instrumentos musicais interativos baseados em aprendizado de máquina.	Fiebrink – Wekinator	ML interativo; regressão e classificação com sensores.
2009	Instrumento puramente digital baseado em gestos.	Kiefer, Collins & Fitzpatrick – Phalanger	Interface digital; rastreamento de mãos via vídeo.
2015	Gerar sequências musicais controlando redundância, hierarquia e estruturas globais.	Tanaka & Fujii	Programação inteira/linear; restrições lineares e hierárquicas.
2017	Base de referência do presente trabalho; geração de música considerando estruturas globais.	Cunha, Subramanian & Herremans	Programação inteira/linear; formulação matemática.
2017	Responder em tempo real ao nível de engajamento do artista.	Tahiroğlu, Vasquez & Kildal – NOISA	Regressão para estimar engajamento; interatividade parcial.
2018	Performance personalizada combinando instrumento físico e digital.	Macionis & Kapur – Sansa	Instrumento aumentado com sensores; Kalimba acústica.
2018	Sistemas que se adaptam gradualmente a uma paisagem sonora externa.	Van Nort – Organismos Geneticamente Sonificados	Processos generativos adaptativos; abordagem ecológica.
2019	Tomada de decisão generativa em tempo real baseada em reconhecimento de gestos.	Agioti	Redes neurais; classificação de gestos em tempo real.

Pesquisadores há muito exploram como o aprendizado de máquina pode facilitar o diálogo entre o artista e o sistema computacional. As primeiras abordagens baseadas em *kits* de ferramentas, como o *Wekinator* de Fiebrink et al. (2009), permitiram que compositores mapeassem entradas de sensores a parâmetros sonoros por meio de regressão e classificação, abrindo caminho para instrumentos como o *Sansa* de Macionis e Kapur (2018) e o *Phalanger* de Kiefer, Collins e Fitzpatrick (2009). Esses dois se destacam por seus diferentes escopos: o *Sansa* utiliza o aumento baseado em sensores de uma *Kalimba* acústica para performances altamente personalizadas, enquanto o *Phalanger* emprega rastreamento de mãos baseado em vídeo em uma interface puramente digital. Com base nesses conceitos, o *NOISA* de Tahiroğlu, Vasquez e Kildal (2017) integra autonomia parcial, usando um algoritmo de regressão para estimar o engajamento do artista e responder de acordo, borrando assim as fronteiras entre instrumentos puramente reativos e interativos. Em contraste, os Organismos Geneticamente Sonificados de Nort (2018) situam os processos generativos em um cenário ecológico mais amplo, "adap-

tando"gradualmente seus vocabulários sintetizados a uma paisagem sonora externa. Tal como no caso do Gioti (2019), que utiliza redes neurais para classificar gestos musicais para tomada de decisões generativas em tempo real, estes exemplos sublinham uma tendência de se incorporar a aprendizagem de máquina em sistemas criativos e interativos.

Esses trabalhos mostram, por meio das variadas técnicas apresentadas, como o campo da música generativa é amplo. Assim, além de contextualizar o presente trabalho, servem de motivação para a investigação e aprofundamento de técnicas já existentes, bem como a experimentação de abordagens alternativas.

4 DEFINIÇÃO DO PROBLEMA E PROPOSTA DE SOLUÇÃO

Nesta seção, o fluxo que envolve a concepção do problema, sua modelagem, resolução e pós-processamento são apresentados. Começa-se com as definições dos elementos que compõem o problema, como os arranjos de notas musicais e as diversas possibilidades de agregação desses arranjos, cada uma apresentando um valor de qualidade musical. Então, modela-se esses arranjos como nodos de um grafo, que contém arcos valorados de acordo com a qualidade musical gerada pela transição de um arranjo (nodo) para outro. Assim, obtém-se um problema de programação linear inteira-mista, no qual deseja-se encontrar o caminho de custo mínimo para um dado número de arranjos que se deseja incluir no solo. Esse problema é resolvido por meio da sua apropriada definição formal, gerando-se uma instância que é resolvida por meio do *solver* Gurobi que implementa o método *branch-and-cut*, apresentado na Seção 2.2.3. Por fim, é gerado um arquivo XML de saída, contendo o solo e que pode ser aberto em *softwares* de edição de música como o *Guitar Pro*.

4.1 ARRANJOS DE NOTAS MUSICAIS

No cenário construído no experimento, há conjuntos de notas musicais, *licks*, que podem ser arranjados de modo a formar frases musicais de tamanho arbitrário. Os arranjos gerados pelos autores Cunha, Subramanian e Herremans (2018) possuem por volta 30 segundos de duração. O desafio é, para um determinado número de *licks* selecionados de um subconjunto arbitrário do conjunto que contém todos os *licks*, produzir o arranjo musicalmente mais adequado. Adequação musical é definida no artigo em questão, baseando-se em uma média ponderada de opiniões de músicos profissionais (maior peso) e de amadores (menor peso) do gênero musical escolhido (*blues*).

4.2 MODELAGEM COMPUTACIONAL PARA COMPOSIÇÃO DE SOLOS

A seguir, a construção do grafo e a formulação do problema de programação linear inteira-mista resultante são apresentados em maior detalhe. Ao fim desse processo, resta apenas o pós-processamento do arquivo gerado para que o resultado possa ser escutado.

4.2.1 Construção do grafo de *licks*

No experimento, utilizou-se um *dataset* contendo arquivos XML, cada um representando um conjunto de notas musicais, *licks*, organizados em pastas que descreviam sua categoria. Para fins de valoração das transições entre *licks* como mostrado na Tabela 2, as categorias relevantes são: regulares, de repetição, *turnaround* e que começam ou terminam com uma pausa. Todavia, a Tabela 3 mostra que *licks* regulares, de repetição e que começam ou terminam

com uma pausa podem ser tanto rápidos, moderados ou devagar, enquanto *licks* de *turnaround* podem ser regulares ou começarem com uma pausa.

Tabela 2 – Com base na natureza de um par de *licks* L1 e L2 quaisquer, é definido o custo de transição de L1 para L2. Este custo de transição pode ser do tipo bônus ou penalidade e cada tipo de transição está associado a uma preferência T1, T2, ... T8. (Cunha et al., 2017)

Preferência	Transição	Tipo	Custo
T1	De um <i>lick</i> de repetição para qualquer outro <i>lick</i>	Bônus	-50
T2	De um <i>lick</i> que não termina com pausa para qualquer outro que comece com pausa ≤ 1 batida	Bônus	-15
T3	De um <i>lick</i> que termina com uma pausa ≤ 1 batida para qualquer outro que comece com pausa	Bônus	-15
T4	De qualquer <i>lick</i> i para qualquer <i>lick</i> j em que a primeira nota de j é adjacente e está na mesma oitava da última nota de i na escala pentatônica maior/menor (ex.: i = C5 e j = D5)	Bônus	-15
T5	De qualquer <i>lick</i> i para qualquer <i>lick</i> j em que a primeira nota de j é igual à última nota de i	Bônus	-15
T6	De qualquer <i>lick</i> que termina com uma pausa > 1 batida para qualquer outro <i>lick</i> que começa com uma pausa > 1 batida	Penalidade	+25
T7	De qualquer <i>lick</i> que termina com uma pausa > 2 batidas para qualquer outro <i>lick</i>	Penalidade	+25
T8	De qualquer <i>lick</i> para qualquer outro que começa com uma pausa > 2 batidas	Penalidade	+25

Para se observar o problema da devida maneira que possibilita a aplicação de otimi-

zações, foi construído um grafo a partir de um subconjunto arbitrário do conjunto que contém todos os *licks*, no qual cada *lick* foi tratado como um nodo e, a transição entre um *lick* e outro obrigatoriamente diferente, como um arco. A fim de se atingir esse objetivo, gerou-se uma matriz de custos que representa o grafo. Para que a matriz de custos seja gerada, precisou-se atribuir valores aos arcos do grafo e cada arco foi valorado com base nas classificações dos dois nodos entre eles, sendo que esta valoração foi feita de maneira ponderada, com opiniões de músicos profissionais (maior peso) e de amadores (menor peso). Assim, segue-se a Tabela 2 para se preencher cada célula da matriz de custos.

Porém, percebe-se que antes deve-se consultar a classificação de cada nodo (*lick*) do grafo para se obter os valores de custo de cada arco (cada transição possível entre *licks*). Desse modo, consulta-se a Tabela 3.

Todavia, o *dataset* não contém arquivos XML (*licks*) classificados da maneira como mostra a Tabela 3. Então, um tratamento do *dataset* foi elaborado manualmente de modo que as classificações necessárias dos *licks* fossem feitas. Esse tratamento manual tornou o *dataset* utilizado neste trabalho ligeiramente diferente do *dataset* dos autores do artigo de referência.

Após a geração da matriz de custos para o subconjunto arbitrário do conjunto que contém todos os *licks*, podemos visualizar o grafo gerado, como no exemplo da Figura 8.

Na Figura 8, o grafo gerado contém nodos (*licks*) pertencentes a quatro categorias: regulares, de repetição, *turnaround* e que começam ou terminam com uma pausa. Além disso, notamos que há um caminho traçado no grafo. Este conjunto de arcos que compõem o caminho representam a solução e são escolhidos de modo que, para um dado número de nodos que queremos incluir no caminho solução, sejam escolhidos os arcos cuja soma dos seus custos seja a mínima possível. Desse modo, estamos gerando, para um dado número de nodos que queremos incluir no caminho solução, o conjunto de nodos (*licks*) que corresponde ao melhor arranjo de conjunto de notas musicais dentre os possíveis de serem selecionados no dado subconjunto arbitrário do conjunto que contém todos os *licks*. Assim, percebemos como o problema é modelado como uma variante ESPPRC, apresentado na Seção 2.3.

Tabela 3 – Categorias que cada *lick* pode apresentar e as quantidades que cada categoria de *lick* está presente no *dataset*. (Cunha et al., 2017)

	Regular	Com pausa			Repetição	Total
		Início	Fim	Ambos		
<i>Fast blues</i>	33	15	13	3	1	65
<i>Moderate blues</i>	93	46	53	5	19	216
<i>Slow blues</i>	14	18	9	5	3	49
<i>Turnaround licks</i>	10	2	0	0	0	12
Total	150	81	75	13	23	342

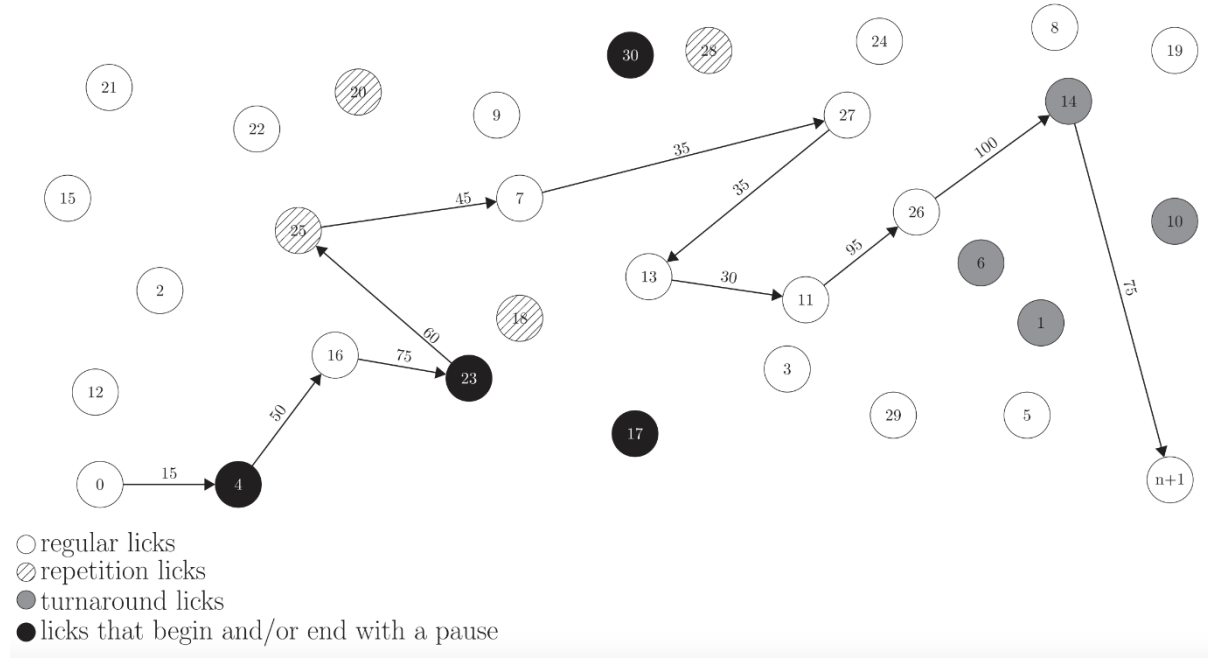


Figura 8 – Um caminho contendo 10 nodos, mais os nodos inicial e final, de modo que, dentre todos os caminhos possíveis de 10 nodos, este seja um dos caminhos que apresentam o custo mínimo. Nodos brancos são regulares, listrados são de repetição, cinza são de *turnaround* e pretos são os que começam e/ou terminam com uma pausa. (Cunha et al., 2017).

Seja $G = (L, A)$ um grafo dirigido, sendo L o conjunto de $n + 2$ vértices (inclui os vértices 0 e $n + 1$), $L' = L \setminus \{0, n + 1\}$ e $A = \{(i, j) | i \neq j\}$ o conjunto de arcos, G contém vértices, cada vértice podendo pertencer a uma de quatro categorias, regulares, de repetição, *turnaround* e que começam ou terminam com uma pausa e cada arco (i, j) conecta um vértice $i \in L$ a um vértice $j \in L$ e apresenta um custo de transição p_{ij} , obtido ao se consultar a tabela 1. Os parâmetros b , r e s representam a quantidade de compassos que deve estar presente no solo solução, a quantidade máxima de *licks* de repetição que deve estar presente no solo solução e a quantidade máxima de *licks* de que começam ou terminam com uma pausa que deve estar presente no solo solução, respectivamente. A duração de cada *lick* $i \in L$ é c_i , expressa em quantidade de compassos. A variável y_i vale 1 se o *lick* $i \in L'$ está presente no solo solução e 0 caso contrário. A escolha dos vértices 0 e $n + 1$ é feita previamente ao cálculo para se encontrar o caminho de custo mínimo, de modo que estes vértices devem pertencer à categoria regular. Define-se T e R como o conjunto de *licks* de *turnaround* e repetição, respectivamente, e P como o conjunto de *licks* que começam e/ou terminam com uma pausa.

4.2.2 Formulação do problema de programação linear inteira-mista

A partir do grafo de *licks* gerado, o objetivo é a escolha de vértices de modo a minimizar $\sum_{(i,j) \in A} p_{ij} x_{ij}$, onde x_{ij} vale 1 se o arco for escolhido na solução e 0, caso contrário. Além disso, deve-se seguir algumas restrições de modo a gerar a solução que contenha o caminho de custo mínimo.

Neste estágio, considera-se a formulação do problema como um problema de programação linear inteira-mista da seguinte maneira (CUNHA; SUBRAMANIAN; HERREMANS, 2018).

$$\min \sum_{(i,j) \in A} p_{ij} x_{ij} \quad (1)$$

$$\text{s.t.} \quad \sum_{j \in L \setminus \{0\}} x_{ij} = y_i \quad \forall i \in L' \quad (2)$$

$$\sum_{i \in L \setminus \{n+1\}} x_{ij} = y_j \quad \forall j \in L' \quad (3)$$

$$\sum_{i \in L'} c_i y_i = b \quad (4)$$

$$\sum_{i \in R} y_i \leq r \quad (5)$$

$$\sum_{i \in P} y_i \leq s \quad (6)$$

$$\sum_{j \in L' \setminus T} x_{0j} = \sum_{i \in T} x_{i,n+1} = 1 \quad (7)$$

$$x_{ij} + x_{ji} \leq 1 \quad \forall (i, j) \in A, i < j \quad (8)$$

$$\sum_{i \in S} \sum_{j \in \bar{S}} x_{ij} \geq 1 \quad \forall S \subseteq L', 2 < |S| < b \quad (9)$$

$$x_{ij} \in \{0, 1\} \quad \forall (i, j) \in A \quad (10)$$

$$y_i \in \{0, 1\} \quad \forall i \in L' \quad (11)$$

A função objetivo (1) minimiza os custos de transição entre os *licks*. As restrições (2) e (3) estabelecem, respectivamente, que exatamente um arco deve deixar e chegar a um *lick* selecionado para o solo, garantindo assim a continuidade do solo e que cada *lick* ocorra no máximo uma vez. A restrição (4) impõe que a duração total dos *licks* (em compassos) usados no solo deve ser igual a b . A restrição (5) limita o solo a ter no máximo r *licks* de repetição. A restrição (6) garante que o solo tenha no máximo s *licks* que começam ou terminam com pausa. A restrição (7) garante que o solo comece com um *lick* diferente do tipo *turnaround* e termine com um do tipo *turnaround*. As restrições (8) e (9) são as chamadas restrições de subciclos, que na prática previnem a ocorrência de solos múltiplos e desconexos. Essa classe de restrições é bem conhecida na literatura sobre o problema do caixeiro viajante (TSP) (APPLEGATE et al., 2011). Como há um número exponencial de restrições (9), ou seja, uma para cada subconjunto possível S no conjunto L , elas são adicionadas sob demanda, iterativamente (CHEN; BATSON; DANG, 2011). Observe que o número de restrições pode ser limitado considerando apenas os subconjuntos que estão dentro do intervalo $2 < |S| < b$. Isso ocorre porque o número de *licks* no solo nunca excederá o valor de b e, portanto, nunca haverá subciclos maiores que b , e o caso particular em que $|S|=2$ já é tratado pela restrição (8). Finalmente, as restrições (10) e (11) definem o domínio das variáveis de decisão.

Quando as restrições (5) e (6) são relaxadas, o problema pode ser visto como uma variante do ESPPRC apresentado na seção 2.3. Este mesmo comentário é feito por Cunha, Subramanian e Herremans (2018).

Após modelado o problema de programação linear inteira-mista, cada instância pode ser resolvida por meio do método *branch-and-cut*, apresentado na seção 2.2.4.

4.2.3 Pós-processamento

O solo gerado pelo processo de otimização apresenta o formato XML, que pode ser lido por *softwares* de edição musical que geram a partitura correspondente e tocam a música escrita, como *Guitar Pro*. Uma alternativa gratuita e sem a necessidade de instalação de programa algum é o *Guitar Tab Pro*.

5 RESULTADOS

Considerando-se o número de *licks* selecionados do *dataset* do artigo Cunha, Subramanian e Herremans (2018) mostrado no exemplo da Figura 8 para a geração de solos, foram feitas comparações de desempenho para diferentes parâmetros utilizados no programa que implementa a modelagem do problema de programação linear inteira-mista formulado na Seção 4.2.2. Utilizou-se uma máquina com uma CPU Apple M2 de 8 núcleos, 8 GB de RAM e GPU de 10 núcleos.

5.1 LICKS DOS TIPOS SLOW, MODERATE E FAST

Na Tabela 4, são mostradas médias e medianas dos valores das soluções ótimas encontradas, do tempo gasto pelo modelo para se atingir uma solução ótima e do número de subciclos encontrados antes de se determinar uma solução ótima. L representa o número de *licks* possíveis de serem escolhidos para compor a solução ótima (parâmetro b foi considerado 12 para todos estes casos de teste, ou seja, o número de compassos e, consequentemente, o número de *licks* que devem estar presentes em cada uma das soluções ótimas foi constante). Para cada valor de L , são sorteados L *licks* dentre todos os existentes daquela categoria no dataset e esse processo é repetido 100 vezes, obtendo-se a média e a mediana do tempo para se atingir a solução ótima, número de ciclos encontrados até se encontrar a solução ótima e qualidade da solução ótima (quanto menor, maior a qualidade).

Tabela 4 – Médias e medianas dos valores das soluções ótimas encontradas (quanto menor, maior a qualidade), do tempo gasto pelo modelo para se atingir uma solução ótima e do número de subciclos encontrados antes de se determinar uma solução ótima. L representa o número de *licks* possíveis de serem escolhidos para compor a solução ótima (parâmetro b foi considerado 12 para todos estes casos de teste, ou seja, o número de compassos e, consequentemente, o número de *licks* que devem estar presentes em cada uma das soluções ótimas foi constante).

Perfil	L	Tempo (s)		Número de subciclos		Qualidade	
		Média	Mediana	Média	Mediana	Média	Mediana
Slow	32	0,28	0,19	1,17	1,00	-40,85	-40,00
	43	0,65	0,36	1,43	1,00	-75,45	-80,00
Moderate	32	2,26	0,20	5,43	1,00	-75,40	-70,00
	62	10,55	0,85	5,73	1,00	-181,80	-190,00
	160	10,68	6,34	2,11	1,00	-248,70	-250,00
Fast	32	0,57	0,19	2,01	1,00	-114,45	-110,00
	62	4,74	0,95	5,08	1,00	-193,35	-190,00

Na Tabela 4 são apresentados os valores médios e medianos de tempo, número de subciclos e qualidade das soluções ótimas encontradas pelo modelo. Observa-se que, para instâncias do tipo *Slow*, o tempo de execução é bastante reduzido, com valores médios inferiores a um segundo e medianas ainda menores. Nessas situações, o algoritmo demonstra elevada eficiência, convergindo de forma rápida e consistente para soluções ótimas. Contudo, à medida que o número de *licks* possíveis aumenta, especialmente nos cenários *Moderate* e *Fast*, o tempo médio cresce significativamente, alcançando valores acima de dez segundos em algumas configurações. A discrepância entre as médias e as medianas dos tempos de execução e número de subciclos detectados nesses casos sugere a presença de instâncias mais complexas, que exigem maior esforço computacional, ainda que a maioria das execuções permaneça relativamente rápida.

Em relação ao número de subciclos, verifica-se que a mediana é igual a 1 em todos os experimentos, indicando que, para um valor de L e uma categoria escolhida, após encontrarmos uma solução ótima para cada uma das 100 vezes em que escolhemos aleatoriamente $b - 1$ *licks* dentre os L disponíveis da categoria considerada, eliminamos apenas uma quantidade de 0, 1 ou outro valor maior do que 1 de subciclos para a obtenção da solução ótima. Ainda, as médias mais elevadas revelam que, em algumas instâncias, sobretudo nas configurações *Moderate*, a quantidade de subciclos que devem ser eliminados para se encontrar a solução ótima é consideravelmente maior do que as outras quantidades observadas (0 ou 1). Esse comportamento aponta para certa instabilidade, uma vez que, embora o algoritmo costuma convergir rapidamente, existem casos pontuais que demandam maior número de iterações para alcançar a solução ótima.

No que se refere à qualidade das soluções, observa-se uma tendência clara: quanto maior o valor de L , melhores são as soluções obtidas. Nos cenários *Slow*, os valores de qualidade permanecem relativamente modestos, mas em *Moderate* e *Fast* há uma melhoria expressiva, destacando-se a configuração *Moderate* com $L = 160$, que apresentou as soluções de melhor qualidade entre todos os experimentos (com média de -248,7 e mediana de -250). Esses resultados evidenciam um *trade-off* importante: embora instâncias maiores impliquem maior custo computacional e variabilidade no tempo de execução, elas proporcionam soluções de qualidade significativamente superior.

De forma geral, os resultados apontam que o modelo é eficiente em instâncias pequenas, mas seu verdadeiro potencial é revelado em instâncias de maior porte, quando se busca maximizar a qualidade das soluções. A escolha da configuração ideal, portanto, depende do equilíbrio desejado entre tempo de execução e qualidade da solução, sendo que as instâncias *Moderate* se mostram particularmente vantajosas quando a prioridade está na obtenção de soluções mais robustas.

5.2 ANÁLISE QUALITATIVA

Em termos qualitativos, ou seja, a qualidade sonora dos trechos musicais gerados, observou-se divergências em relação aos trechos musicais gerados pelos autores do artigo de referência. Notou-se, do ponto de vista do autor, uma qualidade musical aceitável, porém não superior à qualidade dos solos gerados pelos autores do artigo de referência. Entretanto, uma coleta de opiniões de músicos profissionais e amadores deveria ser feita a fim da obtenção de uma avaliação mais robusta dos resultados gerados.

Em termos de potencial para a geração de uma obra musical completa e não apenas trechos, cada solução encontrada pelo programa resolvidor, isto é, cada trecho musical gerado, pode ser concatenado com um novo trecho musical, de modo a compor uma faixa musical completa após atingida uma certa quantidade de trechos musicais concatenados. Além disso, também é possível a geração de uma obra musical de duração bastante longa, dado o número explosivo de combinações de *licks* e o tempo para se obter um novo trecho musical, condições que possibilitam a geração de novos trechos musicais enquanto os já gerados estão sendo tocados.

6 CONCLUSÃO

Após a reprodução do experimento realizado por Cunha, Subramanian e Herremans (2018), foram gerados, a partir do *dataset* fornecido pelos autores, conjuntos de *licks* que apresentam a mesma quantidade de *licks* que os conjuntos de *licks* gerados pelo experimento realizado pelos autores. Quando comparados os áudios dos arquivos XML relativos aos conjuntos de *licks* gerados, há diferenças musicais, cuja qualidade deve ser futuramente avaliada por músicos profissionais e amadores, semelhantemente à avaliação dos áudios gerados pelo experimento realizado pelos autores.

Houve um tratamento manual para a organização do *dataset* fornecido pelos autores para possibilitar a utilização deste na implementação do programa elaborado para a resolução do problema de otimização. Essas questões são referentes à fidelidade da modelagem em relação à modelagem de referência elaborada pelos autores. Resultados sonoros divergentes eram esperados e foram observados por conta desses aspectos.

Com respeito à eficiência e à natureza do programa resolvidor elaborado, que foram expostas na Seção 5, observaram-se resultados positivos no quesito de tempo de execução evidentemente aceitável para todas as instâncias avaliadas, tornando a utilização do programa viável para instâncias de diferentes tamanhos, dentro dos limites de tamanho impostos pela dimensão do *dataset*. Também foi visto que, conforme as instâncias crescem em tamanho, melhora a qualidade da solução encontrada.

Portanto, após a reprodução dos experimentos de Cunha, Subramanian e Herremans (2018), obteve-se resultados sonoros divergentes dos obtidos pelo experimento original e o programa elaborado objetivando a solução de instâncias do problema modelado mostrou-se eficiente no quesito de tempo de solução para a obtenção de soluções ótimas. Para trabalhos futuros, pode-se considerar métodos de solução alternativos para o problema de otimização, tais como aprendizado por reforço, *simulated annealing* e colônia de formigas.

REFERÊNCIAS

- APPLEGATE, D. L. et al. The traveling salesman problem: a computational study. In: **The Traveling Salesman Problem**. [S.l.]: Princeton university press, 2011.
- ARENALES, M. N. et al. Pesquisa operacional. 2007.
- CHEN, D.-S.; BATSON, R. G.; DANG, Y. **Applied integer programming: modeling and solution**. [S.l.]: John Wiley & Sons, 2011.
- COPE, D. Experiments in musical intelligence. (**No Title**), 1996.
- CUNHA, N. d. S.; SUBRAMANIAN, A.; HERREMANS, D. Generating guitar solos by integer programming. **Journal of the Operational Research Society**, Taylor & Francis, v. 69, n. 6, p. 971–985, 2018.
- FEILLET, D. et al. An exact algorithm for the elementary shortest path problem with resource constraints: Application to some vehicle routing problems. **Networks**, v. 44, n. 3, p. 216–229, out. 2004.
- FIEBRINK, R. et al. A meta-instrument for interactive, on-the-fly machine learning. 2009.
- GENDREAU, M.; POTVIN, J.-Y. et al. **Handbook of metaheuristics**. [S.l.]: Springer, 2010. v. 2.
- GIOTI, A.-M. Imitation game: Real-time decision-making in an inter-active composition for human and robotic percussionist. In: **Proceedings of the International Computer Music Conference**. [S.l.: s.n.], 2019.
- HERREMANS, D.; CHUAN, C.-H.; CHEW, E. A functional taxonomy of music generation systems. **ACM Computing Surveys**, v. 50, n. 5, p. 69:1–69:30, 2017.
- JI, S.; YANG, X.; LUO, J. A survey on deep learning for symbolic music generation: Representations, algorithms, evaluations, and challenges. **ACM Computing Surveys**, v. 56, n. 1, p. 7:1–7:39, 2023.
- KIEFER, C.; COLLINS, N.; FITZPATRICK, G. Phalanger: Controlling music software with hand movement using a computer vision and machine learning approach. University of Sussex, 2009.
- MACIONIS, M. J.; KAPUR, A. Sansa: A modified sansula for extended compositional techniques using machine learning. In: **Proceedings of the International Conference on New Interfaces for Musical Expression**. [S.l.: s.n.], 2018. p. 78–81.
- MIRANDA, E. R. **Composing Music with Computers**. New York: Routledge, 2001.
- NIERHAUS, G. **Algorithmic Composition: Paradigms of Automated Music Generation**. [S.l.]: Springer Vienna, 2009.
- NORT, D. V. Genetically sonified organisms: Environmental listening/sounding agents. In: **Proceedings of the Musical Metacreation Workshop**. [S.l.: s.n.], 2018.
- PACHET, F. The continuator: Musical interaction with style. **Journal of New Music Research**, Taylor & Francis, v. 32, n. 3, p. 333–341, 2003.

TAHIROĞLU, K.; VASQUEZ, J. C.; KILDAL, J. Facilitating the musician's engagement with new musical interfaces: Counteractions in music performance. **Computer Music Journal**, MIT Press, v. 41, n. 2, p. 69–82, 2017.

TANAKA, T.; FUJII, K. Describing global musical structures by integer programming on musical patterns. In: SPRINGER. **International Conference on Mathematics and Computation in Music**. [S.l.], 2015. p. 52–63.

WOLSEY, L. A.; NEMHAUSER, G. L. **Integer and combinatorial optimization**. [S.l.]: John Wiley & Sons, 1999.

ZHU, Y. et al. A survey of ai music generation tools and models. **arXiv preprint arXiv:2308.12982**, 2023.

A RESUMO EM FORMATO DE ARTIGO DA SBC

Geração de Solos de Guitarra como um Problema de Otimização

Aluno: Sergio Bonini, Orientador: Pedro Belin Castellucci

Universidade Federal de Santa Catarina – Curso de Ciência da Computação
Florianópolis – SC – Brasil

RESUMO

Neste trabalho, a geração musical automática foi explorada, especificamente para a geração de solos de guitarra. A atividade foi tratada como um problema de otimização no qual um aglomerado de conjuntos de notas musicais foi agrupado a fim de se gerar a melhor combinação possível em termos de qualidade musical, baseando-se em opiniões previamente coletadas de músicos profissionais. Para tal objetivo, utilizou-se um artigo que contém o *dataset* com os conjuntos de notas musicais explorados, bem como os critérios para a atribuição de valores que indicam a qualidade musical para cada transição entre estes conjuntos. Ainda com base nesse artigo, a formulação do problema de otimização em termos de programação inteira foi elaborada e utilizou-se a biblioteca PuLP da linguagem Python em conjunto com o *solver* Gurobi. Em seguida, uma análise do programa solução e de seu desempenho foi realizada, e soluções alternativas que podem ser avaliadas em trabalhos futuros foram mencionadas. Comparações entre os conjuntos de notas musicais gerados no experimento e no artigo base foram feitas, além da análise do impacto de diferentes parâmetros no custo computacional para a solução do modelo. Por fim, observaram-se resultados sonoros divergentes dos obtidos no artigo de referência, mas positivos no quesito tempo de execução para a obtenção das soluções ótimas para todas as instâncias avaliadas, tornando a utilização do programa viável para instâncias de diferentes tamanhos, dentro dos limites impostos pela dimensão do *dataset*.

ABSTRACT

In this work, automatic music generation was explored, specifically for the generation of guitar solos. The activity was treated as an optimization problem in which a cluster of musical note sets was grouped in order to generate the best possible combination in terms of musical quality, based on previously collected opinions from professional musicians. For this purpose, an article containing the dataset with the explored musical note sets was used, as well as the criteria for assigning values, which indicate the musical quality for each transition between these sets. Also based on the article, the formulation of the optimization problem in terms of integer programming was elaborated and the PuLP library of the Python language was used, with the CBC solver. Then, an analysis of the solution program and its performance was carried out and alternative solutions that can be evaluated in future works were cited. Comparisons between the musical note sets generated in the experiment and in the base article were made, in addition to the impact of different parameters on the computational cost for the model solution. Finally, the results observed differed from those obtained in the reference article, but were positive in terms of execution time for obtaining optimal solutions for all evaluated instances, making the use of the program viable for instances of different sizes, within the size limits imposed by the dataset dimension.

A.1 INTRODUÇÃO

A geração musical por meios computacionais vem sendo explorada a partir de múltiplas perspectivas, incluindo composição algorítmica, aprendizado de máquina e sistemas iterativos (NIERHAUS, 2009; MIRANDA, 2001; ZHU et al., 2023). Levantamentos recentes destacam a diversidade de tarefas (composição, acompanhamento, harmonização, improvisação) e de representações e modelos utilizados (HERREMANS; CHUAN; CHEW, 2017; JI; YANG; LUO, 2023). Nesse cenário, técnicas de otimização combinatória, em especial baseadas em programação matemática, têm se mostrado adequadas para incorporar regras estruturais e estilísticas à geração musical (TANAKA; FUJII, 2015).

Entre as vertentes da chamada música generativa, destaca-se o desafio de gerar trechos musicalmente coerentes, como solos de guitarra, a partir de critérios objetivos que reflitam preferências humanas. Cunha et al. (CUNHA; SUBRAMANIAN; HERREMANS, 2018) propuseram um modelo de programação inteira-mista para gerar solos no estilo *blues* de 12 compassos, combinando *licks* pré-existentes de acordo com uma matriz de custos baseada em avaliações de músicos.

Inspirado por esse trabalho, o presente artigo tem como objetivo principal analisar e reproduzir o experimento de Cunha, Subramanian e Herremans (2018), adotando a mesma ideia central: formular a geração de solos como um problema de otimização em grafos, em que *licks* são nós conectados por transições ponderadas. Em relação ao trabalho original, buscamos:

- detalhar a modelagem computacional, desde o tratamento do *dataset* até a construção do modelo inteiro;
- executar um estudo computacional sistemático variando o número de licks candidatos;
- discutir o comportamento do método em termos de tempo de solução, número de subciclos removidos e qualidade das soluções;
- apresentar uma análise qualitativa dos solos gerados a partir de audição informal.

Diferentemente do texto completo do trabalho de conclusão de curso, este artigo omite a fundamentação teórica detalhada sobre teoria musical, TSP, ESPPRC e métodos exatos, concentrando-se na descrição da modelagem, dos experimentos e dos resultados.

O artigo está organizado como segue. Na Seção A.2, apresentamos um panorama sucinto de trabalhos relacionados em música generativa. A Seção A.3 descreve o problema, a modelagem em grafo e a formulação de programação inteira-mista, bem como o fluxo de implementação. A Seção A.4 apresenta os resultados experimentais e a análise qualitativa. Por fim, a Seção A.5 discute conclusões e trabalhos futuros.

A.2 TRABALHOS RELACIONADOS

A literatura em música generativa abrange desde abordagens simbólicas baseadas em regras até modelos estatísticos e de aprendizado profundo (HERREMANS; CHUAN; CHEW, 2017; ZHU et al., 2023). Do ponto de vista histórico, Cope (COPE, 1996) desenvolveu sistemas que analisam obras de um compositor, extraem fragmentos recorrentes e constroem uma gramática composicional capaz de gerar novas peças no estilo original.

Na direção de sistemas interativos, o Continuator de Pachet (PACHET, 2003) combina modelos de Markov de ordem variável com uma interface em tempo real, permitindo que o sistema responda com continuações estilisticamente coerentes ao que o músico toca. Outros trabalhos exploram o uso de *toolkits* de aprendizado de máquina para mapeamento de gestos e sensores a parâmetros sonoros, como o Wekinator de Fiebrink (FIEBRINK et al., 2009), que inspirou instrumentos como o Sansa (MACIONIS; KAPUR, 2018) (Kalimba aumentada com sensores) e o Phalanger (KIEFER; COLLINS; FITZPATRICK, 2009) (instrumento puramente digital baseado em rastreamento de mãos).

Sistemas mais recentes exploram graus variados de autonomia e adaptação. O NOISA (TAHIROĞLU; VASQUEZ; KILDAL, 2017) usa regressão para estimar o engajamento do artista e modular o comportamento do sistema em tempo real. Os Organismos Geneticamente Sonificados, de Van Nort (NORT, 2018), adaptam seus processos generativos a uma paisagem sonora externa. Gioti (GIOTI, 2019) emprega redes neurais para classificar gestos musicais e tomar decisões generativas em tempo real.

Quanto à aplicação de programação inteira e linear, Tanaka e Fujii (TANAKA; FUJII, 2015) modelam a geração de sequências musicais controlando propriedades globais como

hierarquia e redundância por meio de restrições lineares. Nesse contexto se insere o trabalho de Cunha et al. (CUNHA; SUBRAMANIAN; HERREMANS, 2018), que formula a geração de solos de guitarra como um problema de caminho de custo mínimo com restrições estruturais, baseando-se em uma matriz de custos derivada de avaliações subjetivas de transições entre licks.

O presente artigo posiciona-se nessa última vertente, replicando e analisando em detalhes a abordagem de programação inteira-mista proposta em Cunha, Subramanian e Herremans (2018).

A.3 DEFINIÇÃO DO PROBLEMA E ABORDAGEM PROPOSTA

Nesta seção descrevemos a estrutura do *dataset*, a construção do grafo de licks e a formulação do problema como um modelo de programação linear inteira-mista (PLIM). Também apresentamos brevemente o fluxo de implementação e o pós-processamento dos solos gerados.

A.3.1 Arranjos de notas musicais e *dataset*

O ponto de partida é um *dataset* de licks de guitarra no estilo *blues*, fornecido em arquivos XML, conforme o experimento original de Cunha, Subramanian e Herremans (2018). Cada arquivo representa um lick, isto é, um pequeno fragmento melódico que pode ser reutilizado em diferentes solos. Os licks presentes no conjunto de dados estão em tom de Dó, o que garante que os solos gerados permaneçam coerentes em termos de tonalidade.

Para a modelagem, os licks são classificados em categorias de acordo com características musicais relevantes: licks regulares, de repetição, de *turnaround* e licks que começam ou terminam com pausa. Essas categorias são utilizadas para definir a qualidade das transições entre pares de licks e para impor restrições estruturais ao solo gerado (por exemplo, que o solo termine com um lick de *turnaround*).

O *dataset* original não está organizado exatamente nessa taxonomia, o que exigiu um tratamento manual para rotular adequadamente cada lick. Esse pré-processamento é uma das fontes de divergência em relação ao experimento original, pois pequenas diferenças na classificação podem impactar as transições permitidas e, consequentemente, os solos resultantes.

A qualidade musical de uma transição entre dois licks é fornecida por uma matriz de custos construída a partir de avaliações de músicos profissionais e amadores, como descrito em Cunha, Subramanian e Herremans (2018). Profissionais recebem peso maior na média ponderada, refletindo sua maior experiência no estilo.

A.3.2 Modelagem em grafo e formulação em PLIM

Dado um subconjunto de licks selecionado do *dataset*, construímos um grafo dirigido $G = (L, A)$, em que L é o conjunto de vértices (licks) e A é o conjunto de arcos, contendo um

arco (i, j) sempre que é permitido transitar do lick i para o lick j ($i \neq j$). A cada arco (i, j) associamos um custo p_{ij} , obtido a partir da matriz de avaliações subjetivas. Custos menores indicam transições musicalmente mais desejáveis.

Seja $L' = L \setminus \{0, n+1\}$, em que 0 e $n+1$ representam, respectivamente, licks especiais de início e fim do solo, escolhidos previamente e pertencentes à categoria regular. Cada lick $i \in L$ possui uma duração c_i em compassos. O número total de compassos desejado para o solo é dado por b , e os parâmetros r e s delimitam, respectivamente, o número máximo de licks de repetição e de licks que começam ou terminam com pausa na solução. Denotamos por T , R e P os subconjuntos de licks de *turnaround*, de repetição e de pausa.

Introduzimos variáveis binárias x_{ij} que valem 1 se o arco (i, j) pertence ao caminho solução e 0 caso contrário, e variáveis y_i que valem 1 se o lick $i \in L'$ é utilizado no solo e 0 caso contrário. Com essa notação, o problema de gerar o solo é formulado como:

$$\min \sum_{(i,j) \in A} p_{ij} x_{ij} \quad (\text{A.1})$$

$$\text{s.a.} \quad \sum_{j \in L' \setminus \{0\}} x_{ij} = y_i \quad \forall i \in L' \quad (\text{A.2})$$

$$\sum_{i \in L' \setminus \{n+1\}} x_{ij} = y_j \quad \forall j \in L' \quad (\text{A.3})$$

$$\sum_{i \in L'} c_i y_i = b \quad (\text{A.4})$$

$$\sum_{i \in R} y_i \leq r \quad (\text{A.5})$$

$$\sum_{i \in P} y_i \leq s \quad (\text{A.6})$$

$$\sum_{j \in L' \setminus T} x_{0j} = \sum_{i \in T} x_{i,n+1} = 1 \quad (\text{A.7})$$

$$x_{ij} + x_{ji} \leq 1 \quad \forall (i, j) \in A, i < j \quad (\text{A.8})$$

$$\sum_{i \in S} \sum_{j \in \bar{S}} x_{ij} \geq 1 \quad \forall S \subseteq L', 2 < |S| < b \quad (\text{A.9})$$

$$x_{ij} \in \{0, 1\} \quad \forall (i, j) \in A \quad (\text{A.10})$$

$$y_i \in \{0, 1\} \quad \forall i \in L' \quad (\text{A.11})$$

A função objetivo (A.1) minimiza o custo total das transições entre licks. As restrições (A.2) e (A.3) garantem que cada lick selecionado tenha exatamente um arco de entrada e um de saída, assegurando que cada lick seja visitado no máximo uma vez e que o solo seja um caminho simples. A restrição (A.4) impõe que a soma das durações dos licks utilizados seja igual ao número total de compassos b , característico, por exemplo, de um *blues* de 12 compassos.

As restrições (A.5) e (A.6) controlam a quantidade máxima de licks de repetição e de licks com pausa, evitando solos excessivamente redundantes ou fragmentados. A restrição (A.7)

assegura que o solo comece com um lick não-classificado como *turnaround* e termine com um lick dessa categoria, respeitando uma convenção estrutural do estilo. As desigualdades (A.8) e as restrições de eliminação de subciclos (A.9) evitam a formação de ciclos desconexos, assegurando que a solução corresponda a um único caminho. Por fim, (A.10) e (A.11) definem o caráter inteiro das variáveis de decisão.

Quando as restrições (A.5) e (A.6) são relaxadas, o problema pode ser visto como uma variante do problema de caminho mais curto elementar com restrições de recursos (ESPPRC), em que o recurso considerado é o número de compassos. Na prática, as restrições de subciclo (A.9) são em número exponencial e são tratadas por geração de cortes sob demanda, dentro de um esquema *branch-and-cut*.

A.3.3 Fluxo de implementação e pós-processamento

O modelo foi implementado em Python utilizando a biblioteca PuLP para construção do PLIM e o *solver* Gurobi, que implementa métodos do tipo *branch-and-cut*. O fluxo geral é:

1. leitura e organização do *dataset* de licks em XML, com rotulação manual nas categorias necessárias;
2. construção da matriz de custos p_{ij} com base nas avaliações de transições entre licks, conforme Cunha, Subramanian e Herremans (2018);
3. geração de instâncias a partir da escolha de um subconjunto de L licks de uma categoria específica;
4. construção e resolução do modelo PLIM correspondente em PuLP/Gurobi;
5. conversão da solução (sequência de licks) para um arquivo XML de saída contendo o solo completo;
6. visualização e audição do solo em *softwares* como Guitar Pro ou ferramentas equivalentes.

O formato XML de saída permite que o usuário visualize a partitura e ouça o resultado sem necessidade de pós-processamento adicional além da importação em um editor musical ou por meio desta página (<https://dmitrypimonov.com/en/tools/guitar-pro-player>).

A.4 RESULTADOS EXPERIMENTAIS

Os experimentos foram conduzidos em uma máquina com CPU Apple M2 de 8 núcleos, 8 GB de RAM e GPU de 10 núcleos. O número de compassos do solo foi fixado em

$b = 12$ em todas as instâncias, de modo a refletir a forma padrão do *blues* de 12 compassos. Variamos o número de licks disponíveis L em três perfis: *Slow*, *Moderate* e *Fast*, correspondendo a diferentes subconjuntos do *dataset*.

Para cada par (perfil, L), selecionamos aleatoriamente L licks da categoria considerada e repetimos o experimento 100 vezes. Em cada repetição, o modelo gera um solo ótimo (quando encontrado dentro do tempo limite), do qual coletamos: (i) o valor da função objetivo (qualidade, em que valores menores indicam transições melhores), (ii) o tempo de solução e (iii) o número de subciclos eliminados durante o processo de *branch-and-cut*. A Tabela 5 apresenta médias e medianas desses indicadores.

Tabela 5 – Médias e medianas do tempo de solução, número de subciclos e qualidade da solução ótima, para diferentes perfis de licks e valores de L (quanto menor o valor de qualidade, melhor o solo).

Perfil	L	Tempo (s)		Subciclos		Qualidade	
		Média	Mediana	Média	Mediana	Média	Mediana
Slow	32	0,28	0,19	1,17	1,00	-40,85	-40,00
	43	0,65	0,36	1,43	1,00	-75,45	-80,00
Moderate	32	2,26	0,20	5,43	1,00	-75,40	-70,00
	62	10,55	0,85	5,73	1,00	-181,80	-190,00
	160	10,68	6,34	2,11	1,00	-248,70	-250,00
Fast	32	0,57	0,19	2,01	1,00	-114,45	-110,00
	62	4,74	0,95	5,08	1,00	-193,35	-190,00

A.4.1 Discussão dos resultados numéricos

Para instâncias do perfil *Slow*, observa-se que o tempo de execução é bastante reduzido, com médias inferiores a um segundo e medianas ainda menores. Nesses casos, o modelo converge de forma rápida e consistente para soluções ótimas, com número médio de subciclos próximo de 1.

À medida que o número de licks L aumenta, principalmente nos perfis *Moderate* e *Fast*, o tempo médio cresce significativamente, atingindo cerca de 10 segundos para algumas configurações. No entanto, a mediana permanece relativamente baixa, sugerindo que poucas instâncias mais difíceis puxam a média para cima. A mediana do número de subciclos é igual a 1 em todos os casos, mas as médias mais elevadas indicam que, em algumas instâncias, o *solver* precisa eliminar vários subciclos até encontrar o caminho viável de custo mínimo.

Quanto à qualidade das soluções, há uma tendência clara: instâncias com maior L produzem valores médios e medianos da função objetivo mais baixos, indicando solos com transições globalmente mais bem avaliadas. Em particular, o perfil *Moderate* com $L = 160$ apresenta as melhores soluções em termos de qualidade (média de -248,7 e mediana de -250), ao custo de tempos de solução mais elevados.

Esses resultados evidenciam um *trade-off* entre esforço computacional e qualidade musical: instâncias maiores aumentam o espaço de busca e demandam mais processamento, mas proporcionam soluções potencialmente mais ricas e interessantes em termos de combinação de licks.

A.4.2 Análise qualitativa

Além dos indicadores numéricos, realizamos uma análise qualitativa por meio da audição dos solos gerados, comparando-os, quando possível, com os exemplos reportados em Cunha, Subramanian e Herremans (2018). Do ponto de vista subjetivo do autor, os solos produzidos são musicalmente coerentes e reconhecíveis no estilo *blues*, com encadeamentos plausíveis de licks e respeito à forma de 12 compassos.

No entanto, a qualidade percebida não foi superior à dos solos ilustrados no trabalho original. Parte dessa diferença pode ser atribuída ao tratamento manual do *dataset* (reclassificação de licks em categorias) e ao eventual uso de subconjuntos de licks distintos. Pequenas alterações na taxonomia dos licks e na matriz de custos podem levar a variações relevantes do ponto de vista estilístico.

Em termos de potencial, o método se mostra promissor para a geração de obras mais longas. Cada instância do modelo produz um trecho musical (um solo de 12 compassos) que pode ser concatenado com outros trechos gerados em sequência, possibilitando a construção de faixas completas. O tempo de solução obtido nas experiências indica que é factível gerar novos segmentos enquanto solos anteriores estão sendo executados, abrindo caminho para sistemas semi-interativos de improvisação assistida.

A.5 CONCLUSÕES E TRABALHOS FUTUROS

Este artigo apresentou uma reprodução e análise do modelo de programação inteira-mista de Cunha et al. (CUNHA; SUBRAMANIAN; HERREMANS, 2018) para a geração de solos de guitarra no estilo *blues*. Partindo do *dataset* original de licks em formato XML, construímos um grafo de transições ponderadas por avaliações subjetivas de músicos e formulamos um problema de caminho de custo mínimo com restrições estruturais, resolvido por meio de técnicas de *branch-and-cut*.

Os resultados numéricos indicam que o modelo é eficiente para instâncias pequenas e de porte moderado, com tempos de execução compatíveis com aplicações off-line e, em alguns casos, até interativas. Observamos ainda que o aumento do número de licks L tende a melhorar a qualidade da solução, ao custo de maior variabilidade e esforço computacional. A análise qualitativa sugere que os solos gerados são musicalmente aceitáveis, embora diferenças em relação ao experimento original sejam notáveis, em especial devido ao pré-processamento manual do *dataset*.

Como trabalhos futuros, pretendemos:

- realizar uma avaliação sistemática com músicos profissionais e amadores, de forma análoga à usada para construir a matriz de custos, comparando diretamente os solos gerados por diferentes configurações do modelo;
- explorar métodos alternativos de solução, como metaheurísticas (por exemplo, *simulated annealing* e colônia de formigas) e aprendizado por reforço, investigando o compromisso entre optimalidade e tempo de execução;
- enriquecer o modelo com informações adicionais de teoria musical (tensões, resoluções, *voice leading*) e com múltiplas vozes ou camadas;
- investigar o uso do modelo em cenários interativos, nos quais solos são gerados em tempo quase real a partir de decisões do músico em performance.

Em síntese, a combinação de programação inteira-mista com conceitos de teoria musical e avaliação subjetiva de transições mostra-se uma abordagem poderosa e flexível para a geração de solos de guitarra, abrindo caminho para aplicações criativas que unem otimização e prática musical.

B CÓDIGO DESENVOLVIDO

Abaixo, há trechos de código para cada arquivo componente do programa. O repositório pode ser acessado aqui (<https://github.com/Ser9i02021/musicoptimization>).

```

1 from label_chosen_licks import select_N_lick_samples
2 from cost_matrix_construction import build_cost_matrix
3 from optimization import optimize
4 from post_processing import post_process
5 import os
6 import pickle
7
8 def main():
9     # (1) Data Input
10    # (2) Selection of Subset of Licks
11    for i in range(1, 101):
12        filename = f"licks_list_{i}.pkl"
13
14        # Select 30 random licks plus initial and final dummy licks
15        licks_list = select_N_lick_samples(43, 0) # Select 30 random
licks plus initial and final dummy licks
16
17        # If the user has
specified the desired tempo of the
18        # solo as 100 BPM (1:
moderate), then the subset will consist of
19        # random licks taken
from the original database with this same tempo.
20
21    # (3) Calculation of Cost Matrix
p = build_cost_matrix(licks_list)
22
23    # (4) Optimization
graph_path_vertices_ordered,
24    file_paths_for_the_ordered_licks_in_the_solution, obj_val, subt_count
, t_t_ = optimize(licks_list, p, 12)
25
26    with open(filename, 'wb') as f:
27        pickle.dump({
28            'licks_list': licks_list,
29            'graph_path_vertices_ordered':
graph_path_vertices_ordered,
30            'file_paths_for_the_ordered_licks_in_the_solution':
file_paths_for_the_ordered_licks_in_the_solution,
31            'obj_val': obj_val,
32            'subt_count': subt_count,
33            'time_taken': t_t_
34        }, f)
35

```

```

36
37     # (5) Postprocessing
38     # (6) Export as MusicXML
39     output_file = "ordered_licks_optm_output.xml"
40     XML_optm_licks_file = post_process(
file_paths_for_the_ordered_licks_in_the_solution, output_file)
41
42     print(f"Merged MusicXML written to {XML_optm_licks_file}")
43
44
45 def load_chosen_licks_and_obj_val_and_subt_count_from_a_file(file_number
: int) -> list:
46     """
47     Load the chosen licks from the a pickle file.
48     """
49
50     filename = f"licks_list_{file_number}.pkl"
51     if os.path.exists(filename):
52         with open(filename, 'rb') as f:
53             data = pickle.load(f)
54
55     return data['licks_list'], data['graph_path_vertices_ordered'], data
['file_paths_for_the_ordered_licks_in_the_solution'], data['obj_val'
], data['subt_count'], data['time_taken']
56
57 if __name__ == "__main__":
58     main()
59
60     for i in range(1, 101):
61         l, vo, fpo, o, s, t =
load_chosen_licks_and_obj_val_and_subt_count_from_a_file(i)
62         #print(vo, fpo, o, s, t)
63         print(f"File {i}: Objective Value: {o}, Subtour Count: {s}, Time
Taken: {t:.2f} seconds")
64
65 #licks_chosen_from_full_dataset = [l for l, _, _, _, _ in [
load_chosen_licks_and_obj_val_and_subt_count_from_a_file(i) for i in
range(1, 11)]]
66 #print(f"Chosen licks from full dataset: {licks_chosen_from_full_dataset
}")
67
68 #vertices_ordered = [vo for _, vo, _, _, _ in [
load_chosen_licks_and_obj_val_and_subt_count_from_a_file(i) for i in
range(1, 11)]]
69 #print(f"Vertices ordered: {vertices_ordered}")
70
71 #licks_used_in_the_solution = [fpo for _, _, fpo, _, _ in [
load_chosen_licks_and_obj_val_and_subt_count_from_a_file(i) for i in

```

```

    range(1, 11)]]
72 #print(f"Licks used in the solution: {licks_used_in_the_solution}")
73
74 average_obj_val = sum(o for _, _, _, o, _, _ in [
    load_chosen_licks_and_obj_val_and_subt_count_from_a_file(i) for i in
    range(1, 101)]) / 100
75 print(f"Average objective value: {average_obj_val}")
76
77 average_subt_count = sum(s for _, _, _, _, s, _ in [
    load_chosen_licks_and_obj_val_and_subt_count_from_a_file(i) for i in
    range(1, 101)]) / 100
78 print(f"Average number of subtours: {average_subt_count:.2f}")
79
80 average_time_taken = sum(t for _, _, _, _, _, t in [
    load_chosen_licks_and_obj_val_and_subt_count_from_a_file(i) for i in
    range(1, 101)]) / 100
81 print(f"Average time taken: {average_time_taken:.2f} seconds")

```

Listing B.1 – Core training loop from main.py

```

1 import random
2 import os
3 from lxml import etree
4 import numpy as np
5 from random import randint
6
7 # Function to extract pitch information
8 def get_note_info(note):
9     pitch = note.find("pitch")
10    if pitch is not None:
11        step = pitch.find("step").text # Extracts the note name (e.g.,
        F, G)
12        octave = pitch.find("octave").text # Extracts the octave number
13        alter = pitch.find("alter") # Checks for sharp (#) or flat (
        )
14
15        # Convert alter to a string (if applicable)
16        if alter is not None:
17            alter_value = int(alter.text)
18            if alter_value == 1:
19                step += "#" # Sharp note
20            elif alter_value == -1:
21                step += " " # Flat note
22
23        return f"{step}{octave}"
24    else:
25        duration = note.find("duration").text
26        return "Rest w/ the duration (in beats) of " + duration

```

```

27
28 def lick_classification(lick_file_path: str):
29     # Extract information from each lick and classify it from C1 to C9
30     #
    -----
31     # Obtain first and last notes
32     tree = etree.parse(lick_file_path)
33     root = tree.getroot()
34
35     # Find all <note> elements
36     notes = root.findall(".//note")
37     # Extract first and last note if available
38     if notes:
39         first_note = get_note_info(notes[0])
40         last_note = get_note_info(notes[-1])
41         #second_note = get_note_info(notes[1])
42         #print(f"First note: {first_note}")
43         #print(f>Last note: {last_note}")
44         #print(f"Second note: {second_note}")
45     else:
46         print("No notes found in the XML file.")
47
48
49
50     #
    -----
51     # Classify it from C1 to C9
52     lick_classes = []
53
54     all_false = True
55
56     # C1 (Repetition)
57     if lick_file_path.find("repetition") != -1:
58         #print("repetition")
59         lick_classes.append("C1")
60         all_false = False
61
62     # C2 (EwP <= 1)
63     if last_note[0] == "R" and int(last_note[35:]) <= 1:
64         #print("EwP <= 1")
65         lick_classes.append("C2")
66         all_false = False
67
68     # C3 (EwP > 1)
69     if last_note[0] == "R" and int(last_note[35:]) > 1:
70         #print("EwP > 1")

```

```

71         lick_classes.append("C3")
72         all_false = False
73
74     # C4 (Ewp > 2)
75     if last_note[0] == "R" and int(last_note[35:]) > 2:
76         #print("Ewp > 2")
77         lick_classes.append("C4")
78         all_false = False
79
80     # C5 (SwP <= 1)

```

Listing B.2 – Functions for label assignment from label_chosen_licks.py

```

1 import numpy as np
2 from lxml import etree
3 from label_chosen_licks import select_N_lick_samples
4
5
6 def build_cost_matrix(licks_list: list):
7     #licks_list = select_N_lick_samples(num_licks)
8     '''
9     for i in range(len(licks_list)):
10         print(licks_list[i])
11     '''
12     cost_matrix = np.full((len(licks_list), len(licks_list)), 0)
13
14     for i in range(len(licks_list)):
15         for j in range(len(licks_list)):
16             if i != j:
17                 all_false = True
18                 # T1
19                 if "C1" in licks_list[i][2] and ("C1" not in licks_list[j][2]):
20                     cost_matrix[i][j] += -50
21                     #print("cost_matrix[{}][{}]: T1".format(i, j))
22                     all_false = False
23                 # T2
24                 if not (("C2" in licks_list[i][2]) or ("C3" in licks_list[i][2]) or ("C4" in licks_list[i][2])) and ("C2" in licks_list[j][2]):
25                     cost_matrix[i][j] += -15
26                     #print("cost_matrix[{}][{}]: T2".format(i, j))
27                     all_false = False
28                 # T3
29                 if ("C2" in licks_list[i][2]) and (("C5" in licks_list[j][2]) or ("C6" in licks_list[j][2]) or ("C7" in licks_list[j][2])):
30                     cost_matrix[i][j] += -15
31                     #print("cost_matrix[{}][{}]: T3".format(i, j))

```

```

32         all_false = False
33     # T4
34     if licks_list[i][1][0] != "R" and licks_list[j][1][0] !=
35     "R": # Certify that none of the licks are a rest
36         if (((ord(licks_list[i][1][0]) - 2) % 7) == ((ord(
37         licks_list[i][1][0]) - 3) % 7)) and licks_list[i][1][1] ==
38         licks_list[j][1][1]:
39             cost_matrix[i][j] += -15
40             #print("cost_matrix[{}][{}]: T4".format(i, j))
41             all_false = False
42     # T5
43     if licks_list[i][1] == licks_list[j][0]:
44         cost_matrix[i][j] += -15
45         #print("cost_matrix[{}][{}]: T5".format(i, j))
46         all_false = False
47     # T6
48     if "C3" in licks_list[i][2] and "C6" in licks_list[j
49     ][2]:
50         cost_matrix[i][j] += 25
51         #print("cost_matrix[{}][{}]: T6".format(i, j))
52         all_false = False
53     # T7
54     if "C4" in licks_list[i][2] and ("C4" not in licks_list[
55     j][2]):
56         cost_matrix[i][j] += 25
57         #print("cost_matrix[{}][{}]: T7".format(i, j))
58         all_false = False
59     # T8
60     if "C7" in licks_list[j][2]:
61         cost_matrix[i][j] += 25
62         #print("cost_matrix[{}][{}]: T8".format(i, j))
63         all_false = False
64     # T9 (May be removed later. It is being used to identify
65     if the transition does not fit in any of the previous categories)
66     if all_false:
67         cost_matrix[i][j] = 100
68         #print("cost_matrix[{}][{}]: T9".format(i, j))
69     else:
70         cost_matrix[i][j] = 100
71
72     for i in range(len(licks_list)):
73         for j in range(len(licks_list)):
74             print(cost_matrix[i][j], end=" ")
75         print()
76
77     return cost_matrix

```

Listing B.3 – Cost matrix generation from cost_matrix_construction.py

```

1 from label_chosen_licks import select_N_lick_samples
2 from cost_matrix_construction import build_cost_matrix
3
4 from itertools import combinations
5 from pulp import LpProblem, LpVariable, lpSum, LpMinimize, LpBinary
6
7 from pulp import *
8 from collections import defaultdict
9
10 import time
11
12
13
14 def power_set(iterable, min_size, max_size):
15     """
16     Returns the power set (all subsets) of the given iterable
17     as a list of tuples.
18     """
19     s = list(iterable) # Convert to list in case it's a set
20     subsets = []
21     for r in range(min_size, max_size):
22         # combinations(s, r) gives all subsets of length r
23         for combo in combinations(s, r):
24             subsets.append(list(combo))
25     return subsets
26
27
28 '''
29 licks_list = select_N_lick_samples(32) # Select 30 random licks plus
    initial
30                                     # and
    final dummy licks
31 p = build_cost_matrix(licks_list) # Build the cost matrix
32 '''
33 def optimize(licks_list, p, b):
34
35     #L = licks_list # Set of licks (vertices)
36     L = [] # Set of licks (vertices) (0, 1, ..., number_of_licks + 1)
37     for i in range(len(licks_list)):
38         L.append(i)
39
40
41     A = [] # Set of arcs (edges)
42     for i in range(len(L)):
43         for j in range(len(L)):
44             if i != j:
45                 A.append((L[i], L[j]))

```

```

46
47     L_prime = L.copy() # Set of licks (vertices) to be optimized,
without the first and last licks (which are always dummy or regular
licks)
48     for i in range(2):
49         L_prime.remove(L[-i])
50
51     G = (L, A) # Graph G = (L, A) (may not be used)
52
53     R = [] # Set of repetition licks
54     for i in range(1, len(licks_list) - 1):
55         if "C1" in licks_list[i][2]:
56             R.append(i)
57
58     T = [] # Set of turnaround licks
59     for i in range(1, len(licks_list) - 1):
60         if "C8" in licks_list[i][2]:
61             T.append(i)
62
63     P = [] # Set of licks with pause
64     for i in range(1, len(licks_list) - 1):
65         if ("C2" or "C3" or "C4" or "C5" or "C6" or "C7") in licks_list[
i][2]:
66             P.append(i)
67
68
69     # Parameters
70     c = [] # Set of durations (in bars) for each lick
71     for lick in licks_list:
72         c.append(lick[3])
73
74     #b = 11 # Quantity of bars in the solo (will be setted arbitrarily)
75     # Given that there must be exactly one turnaround lick and it is
the
76     # only lick that has 2 bars (others have 1), 'b' will be the
quantity of
77     # nodes (from L_prime) used in the solution plus one
78     r = 1 # Constraint (5) (must be 1)
79     s = 3 # Constraint (6) (must be 3)
80
81
82     #  $2 < |S| < b$  (or  $\text{len}(L\_prime)$  (modified from the paper))
83     #S = power_set(L_prime, 3, len(L_prime)) # # Subsets of L_prime
84
85
86     # Define the optimization problem
87     model = LpProblem("Integer_Programming_Model", LpMinimize)
88

```

```

89 # Define decision variables
90 x = {(i, j): LpVariable(f"x_{i}_{j}", cat=LpBinary) for (i, j) in A}
91 y = {i: LpVariable(f"y_{i}", cat=LpBinary) for i in L_prime}
92
93 # Objective function (1)
94 model += lpSum(p[i][j] * x[i, j] for (i, j) in A)
95
96 # Constraints
97 # Additional constraint to make the modified formulation of (2) and
98 # (3) work
99 model += x[0, L[-1]] == 0
100
101 # (2)
102 '''
103 for i in L_prime:
104     L_minus_zero = L.copy()
105     L_minus_zero.remove(L[0])
106     model += lpSum(x[i, j] for j in L_minus_zero if (i, j) in A) ==
107     y[i]
108 '''
109 for i in L:
110     if i == 0:
111         model += lpSum(x[i, j] for j in L if (i, j) in A) == 1
112     elif i > 0 and i < L[-1]:
113         model += lpSum(x[i, j] for j in L if (i, j) in A) == y[i]
114     else:
115         model += lpSum(x[i, j] for j in L if (i, j) in A) == 0
116
117 '''
118 L_minus_n_plus_1 = L.copy()
119 L_minus_n_plus_1.remove(L[-1])
120 L_minus_n_plus_1.remove(L[0]) # Modification to the formulation on
the paper
for i in L_prime:

```

Listing B.4 – Optimization routine from optimization.py

```

1 import os
2 import xml.etree.ElementTree as ET
3
4 def post_process(input_files, output_file):
5     """
6     Merges multiple MusicXML 'score-partwise' files (assuming each has a
7     single <part>)
8     in the order given by 'input_files'. The measures from the second
9     file onward are
10     appended (with renumbering) to the first file's measures. The merged

```

```

MusicXML is
9  written to 'output_file'.
10
11  :param input_files: List of paths to MusicXML files to be merged
12  :param output_file: Path of the merged output file
13  """
14
15  if not input_files:
16      raise ValueError("No input files provided.")
17
18  # 1) Parse the first file to get the 'base' tree
19  base_tree = ET.parse(input_files[0])
20  base_root = base_tree.getroot()
21
22  # Locate the single <part> in the first file
23  base_part = base_root.find("./part")
24  if base_part is None:
25      raise ValueError(f"No <part> found in first file {input_files
[0]}")
26
27  # Count how many measures currently in the base file
28  measure_offset = len(base_part.findall("measure"))
29
30  # 2) For each subsequent file, parse, renumber, and append
31  for musicxml_path in input_files[1:]:
32      tree = ET.parse(musicxml_path)
33      root = tree.getroot()
34
35      part = root.find("./part")
36      if part is None:
37          raise ValueError(f"No <part> found in file {musicxml_path}")
38
39      measures = part.findall("measure")
40
41      # Renumber measures so they continue after the current offset
42      for idx, measure in enumerate(measures, start=1):
43          measure_number = str(measure_offset + idx)
44          measure.set("number", measure_number)
45
46      # Append the measure to the base part
47      base_part.append(measure)
48
49      # Update measure offset so the next file's measures start after
these
50      measure_offset += len(measures)
51
52  # (3) Write out the merged tree to a new file
53  # ElementTree does not preserve the DOCTYPE, so it won't appear in

```

```

the output.
54     base_tree.write(output_file, encoding="UTF-8", xml_declaration=True)
55
56     with open(output_file, "w", encoding="UTF-8") as f:
57         # Write the xml declaration and doctype manually
58         f.write('<?xml version="1.0" encoding="UTF-8" standalone="no"?>\n')
59         f.write('<!DOCTYPE score-partwise PUBLIC\n')
60         f.write('    "-//Recordare//DTD MusicXML 1.0 Partwise//EN"\n')
61         f.write('    "/musicxml/partwise.dtd">\n')
62
63         # Now let ElementTree write out the merged <score-partwise>
        content
64         base_tree.write(f, encoding="unicode", xml_declaration=False)
65
66     return output_file
67
68
69 '''
70 file_paths_for_the_ordered_licks_in_the_solution = []
71
72 for vertex in graph_path_vertices_ordered:
73     file_paths_for_the_ordered_licks_in_the_solution.append(licks_list[
74         vertex][-1])
75
76 #print(file_paths_for_the_ordered_licks_in_the_solution)
77 '''
78 # Example usage:
79 files_to_merge = [
80     "/home/sergio/Downloads/licks_dataset_sampling/FMS/regular/fast/3.
81     xml",
82     "/home/sergio/Downloads/licks_dataset_sampling/turnaround_with_pause
83     /10.xml",
84     "/home/sergio/Downloads/licks_dataset_sampling/turnaround/12.xml"
85     # ... as many as needed, in the order you want them merged
86 ]
87
88 '''
89 output_file = "ordered_licks_optm_output.xml"
90
91 post_process(file_paths_for_the_ordered_licks_in_the_solution,
92             output_file)
93 print(f"Merged MusicXML written to {output_file}")
94 '''

```

Listing B.5 – Post-processing pipeline from post_processing.py

C EXEMPLOS DE ARQUIVOS DE SAÍDA

Pode-se ouvir o resultado gerado por cada solução encontrada em formato de arquivo XML nesta página (<https://dmitrypimonov.com/en/tools/guitar-pro-player>). Aqui, há exemplos de solução em formato de arquivo XML.

(<https://github.com/Ser9i02021/musicoptimization/tree/main/solutions>)