Trabalho de Conclusão de Curso

AutoTF: Um ambiente Web para modelagem de sistemas físicos

de Marcos Vinícius Santos Costa

orientado por
Prof. Dr. Ícaro Bezerra Queiroz de Araújo

Universidade Federal de Alagoas
Instituto de Computação
Maceió, Alagoas
22 de Abril de 2022

UNIVERSIDADE FEDERAL DE ALAGOAS
Instituto de Computação

AUTOTF: UM AMBIENTE WEB PARA MODELAGEM DE SISTEMAS FÍSICOS

Trabalho de Conclusão de Curso submetido ao Instituto de Computação da Universidade Federal de Alagoas como requisito parcial para a obtenção do grau de Engenheiro de Computação.

Marcos Vinícius Santos Costa

Orientador: Prof. Dr. Ícaro Bezerra Queiroz de Araújo

Banca Avaliadora:

Glauber Rodrigues Leite	Prof. MSC., IC-UFAL
Allan de Medeiros Martins	Prof. Dr., DEE-UFRN

Maceió, Alagoas
22 de Abril de 2022

Catalogação na fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico
Bibliotecária: Taciana Sousa dos Santos – CRB-4 – 2062

C837a Costa, Marcos Vinícius Santos.
AutoTF: um ambiente web para modelagem de sistemas físicos / Marcos Vinícius Santos Costa. – 2022.
109 f. : il. color.

Orientador: Ícaro Bezerra Queiroz de Araújo.
Monografia (Trabalho de Conclusão de Curso em Engenharia da Computação) – Universidade Federal de Alagoas. Instituto de Computação. Maceió, 2022.

Bibliografia: f. 106-109.

1. AutoTF (Sistema web). 2. Modelagem de sistemas. 3. Função de transferência. I. Título.

CDU: 004

UNIVERSIDADE FEDERAL DE ALAGOAS
Instituto de Computação

AUTOTF: UM AMBIENTE WEB PARA MODELAGEM DE SISTEMAS FÍSICOS

Trabalho de Conclusão de Curso submetido ao Instituto de Computação da Universidade Federal de Alagoas como requisito parcial para a obtenção do grau de Engenheiro de Computação.

Aprovado em 22 de Abril de 2022:

Ícaro Bezerra Queiroz de Araújo,
Prof. Dr., Orientador

Glauber Rodrigues Leite,
Prof. MSC., IC-UFAL

Allan de Medeiros Martins,
Prof. Dr., DEE-UFRN

Dedicatória

À minha família

Agradecimentos

Agradeço primeiramente à Deus que me deu energia e sabedoria para concluir este trabalho bem como permitir que eu completasse a graduação.

À minha família, que sempre esteve presente em todas as etapas percorridas até então visando a minha formação. Aos meus pais Marcos e Lucileide, que sempre me apoiaram, me proporcionando o melhor ambiente possível para que eu pudesse desempenhar as minhas atividades, se preocupando sempre nas idas até a universidade após noites mal dormidas, mas entendendo a necessidade de continuar com a rotina puxada para atingir um objetivo maior. À minha irmã Fernanda que sempre se inspirou em mim, me motivando sempre a continuar em busca de conhecimento, e na conclusão do curso.

Agradeço também ao meu orientador Prof. Ícaro Araújo que me propôs a construção deste trabalho. Devido as dificuldades oriundas do cenário pandêmico, o projeto original se tornou inviável. A ideia de um novo projeto que aplicasse a pesquisa em um produto de mercado, que era meu objetivo inicial, foi essencial para manter a motivação em alta.

Aos meus companheiros de turma Alfredo Lima, Lucas Peixoto e Igor Theotônio que me acompanharam nessa jornada desde o início, passando juntos pelas mesmas dificuldades, mas superando como equipe cada uma delas, motivando diariamente um ao outro quando a situação ficava difícil, mas deixando ótimas lembranças e momentos memoráveis que vão ser guardados para muito além da universidade.

Ao meu amigo e colega de trabalho Davyson Omena pela paciência e compreensão da necessidade de tempo que eu precisaria para conseguir concluir este trabalho e fechar minha graduação, garantindo tempo para que eu me ausentasse de minhas obrigações na empresa enquanto elaborava e construía este projeto.

À minha companheira Gabriela Vilalva por entender que precisaria abdicar um pouco do tempo que passamos juntos para que eu finalizasse a graduação, nunca deixando de me apoiar.

Por fim, gostaria de agradecer a todas as pessoas que de alguma forma fizeram parte desta etapa decisiva da minha vida.

22 de Abril de 2022, Maceió - AL

O homem cria a ferramenta. A ferramenta recria o homem.

Marshall McLuhan, Understanding Media: The Extensions of Man (1964)

Resumo

Este trabalho apresenta o AutoTF uma solução desenvolvida para modelagem de sistemas físicos lineares, possibilitando que o usuário visualize a função de transferência à partir de uma variável de entrada e uma variável de saída. Elementos elétricos foram considerados para a construção de uma primeira versão do simulador, abrindo portas para elementos de outras naturezas, visto que cada uma possui componentes que influenciam da mesma forma que fontes, resistores, capacitores ou indutores no sistema ao qual está inserido. A solução foi projetada de modo que a comunicação com o usuário através de sua interface gráfica fosse um diferencial na simulação de circuitos, permitindo a construção de um modelo gráfico a partir da ferramenta Canvas e sua API Javascript, garantindo que através de janelas em um navegador Web pudessem ser vistas todas as etapas até a apresentação do resultado final. O sistema utiliza o grafo de ligação como uma alternativa para representação do modelo proposto pelo usuário, transformando-o em um diagrama de fluxo de sinais proveniente das funções lineares geradas pelas matrizes de interconectividade entre os elementos físicos, para que então a regra de Mason fosse aplicada gerando a função de transferência. Os resultados mostram que o AutoTF atende os objetivos propostos garantindo confiabilidade nas funções resultantes, enquanto recursos gráficos desenvolvidos guiam o projetista no início ao fim do processo de modelagem.

Palavras-chave*: *Modelagem de Sistemas*; *Função de Transferência*; *Sistemas de Controle*; *Regra de Mason.

Abstract

This work presents AutoTF, a solution developed for modeling linear physical systems, allowing the user to visualize the transfer function from an input variable and an output variable. Electrical elements were considered for the construction of a first version of the simulator,creating possibilities to future versions implementations with elements of other natures, since each one has components that influence in the same way as sources, resistors, capacitors or inductors in the system to which it is inserted. The solution was designed so that communication with the user through its graphical interface was a differential in circuit simulation, allowing the construction of a graphical model from the Canvas tool and its javascript API, ensuring that using browser Web pages could be useful to show all the steps until the presentation of the final result. The system uses the link graph as an alternative to represent the model proposed by the user, transforming it into a signal-flow graph from the linear functions generated by the interconnectivity matrices between the physical elements, so that Mason's rule could be applied generating the transfer function. The results show that AutoTF meets the proposed objectives, guaranteeing reliability in the resulting functions, while graphic resources developed guide the designer from the beginning to the end of the modeling process.

Keywords: ***System Modeling***; ***Transfer Function***; ***Control System***; ***Mason's Rule.***

Lista de Figuras

2.1 Exemplo de um grafo simples . . . 21
2.2 Exemplos de grafos dirigidos . . . 22
2.3 Exemplos de árvores . . . 23
2.4 Exemplo de grafo e sua árvore geradora . . . 24

3.1 Sistemas de coordenadas no Canvas HTML5 . . . 28
3.2 Exemplo de um diagrama de classes . . . 32
3.3 Lista duplamente encadeada . . . 34
3.4 Estrutura de filas (FIFO) e pilhas (LIFO) . . . 35
3.5 Estrutura de dados em árvore . . . 35
3.6 Estrutura de dados em grafo . . . 36

4.1 Comportamento das fontes de energia no sistema físico. a) Fonte de Esforço. b) Fonte de fluxo . . . 40
4.2 Relação entre esforço acumulado e fluxo em dispositivos acumuladores . . . 41
4.3 Relação entre fluxo acumulado e esforço em dispositivos acumuladores . . . 42
4.4 Relação entre fluxo acumulado e esforço em dispositivos acumuladores . . . 43
4.5 Exemplos de grafos de ligação . . . 45
4.6 Apresentação de componentes e funcionamento de uma DFS . . . 46
4.7 Exemplo de DFS . . . 47

5.1 Diagrama de Classes dos componentes disponíveis para desenho. . . . 52
5.2 Configurações de rotação disponíveis para os componentes . . . 54
5.3 Diagrama de Classes das conexões . . . 58
5.4 Variáveis personalizadas para definição dos pontos intermediários das conexões . . . 61
5.5 Cenários de colisão conexão-componente. . . . 63
5.6 Tela inicial da aplicação. . . . 67
5.7 Menu lateral da aplicação. . . . 69
5.8 Janela de edição de componente . . . 71
5.9 Ferramentas do menu superior da aplicação. . . . 73
5.10 Mensagem de erro de validação de sistema. . . . 75

5.11 Diagrama de classes da estrutura do grafo de ligação. 78
5.12 Diagrama de classes das variáveis do DFS. 80
5.13 Janela de apresentação de resultados finais 86
5.14 Diagrama de classes dos caminhos do DFS 86

6.1 Sistema 1 sem colisões detectadas à esquerda. Controle de colisão atuando sobre o sistema, à direita 90
6.2 Sistema 2 sem colisões detectadas à esquerda. Controle de colisão atuando sobre o sistema, à direita 91
6.3 Sistema 3 mostrando funcionamento dos controles de colisão entre conexões 92
6.4 Sistema 4 mostrando funcionamento dos controles de colisão evitando que as ligações trombe com outras conexões ou objetos 93
6.5 Janela de apresentação do grafo de ligação do AutoTF acima, DFS abaixo 94
6.6 Janela de apresentação da matriz A1 à esquerda, matriz A à direita 94
6.7 Circuito RLC em série 95
6.8 AutoTF - Resultado para circuito RLC em série 96
6.9 Circuito elétrico com duas malhas 97
6.10 AutoTF - Resultado para circuito elétrico com duas malhas 98
6.11 Circuito elétrico com fonte de corrente e 3 malhas 99
6.12 AutoTF - Resultado para circuito elétrico com fonte de corrente e 3 malhas 101
6.13 Exemplos de circuitos sem solução no AutoTF 102
6.14 Tela de resultados no AutoTF - Cenário sem solução 102

Lista de Tabelas

4.1 Variáveis de esforço e fluxo para cada tipo de sistema 38
4.2 Componentes básicos de uma porta para cada tipo de sistema 39
4.3 Representações matemáticas dos componentes físicos no domínio de Laplace 43

5.1 Representações gráficas dos componentes elétricos 51
5.2 Atributos da classe *Component* 53
5.3 Funções da classe *Component* 54
5.4 Atributos da classe *Connector* 56
5.5 Atributos da classe *Joint* 56
5.6 Atributos da classe *Connection* 57
5.7 Possíveis tipos de conexão entre componentes 60
5.8 Atributos de contexto da etapa de construção do desenho 68
5.9 Atributos da classe *GraphElement* 75
5.10 Novos atributos da classe *GraphComponent* 79
5.11 Atributos da classe *AuxGraphElement* 80
5.12 Preenchimento da matriz A3 para o elemento Girador 82
5.13 Preenchimento da matriz A3 para o elemento Transformador 82

Lista de Símbolos

A Ampére - Unidade de medida para correntes elétricas

$\mathbf{A}$ Matriz resultante de relação de interconectividade entre os componentes de um sistema físico

$\mathbf{A_1}$ Matriz de impedâncias

$\mathbf{A_2}$ Matriz de fontes controladas

$\mathbf{A_3}$ Matriz de transformação e giração

$\mathbf{A_4}$ Matriz de compatibilidade de esforço

$\mathbf{A_5}$ Matriz de continuidade de fluxo

$\mathbf{A_{5N}}$ Matriz de continuidade de fluxo linearmente independente

C Variável de capacitância de um capacitor elétrico

Δ Determinante do DFS

Δ_k K-ésimo cofator do DFS

E Energia envolvida em um componente ou sistema físico

e Variável de esforço

e_a Esforço acumulado

F Faraday - Unidade de medida para capacitâncias

f Variável de fluxo

f_a Fluxo acumulado

G Relação de giração

$G(s)$ Função de transferência

G_k Ganho do k-ésimo caminho direto no DFS

γ Fluxo magnético concatenado

$\phi(\rho)$ Representação de função linear

H Henry - Unidade de medida para indutâncias

i Corrente elétrica

L Variável de indutância de um indutor elétrico

N Relação de transformação

Ω Ohm - Unidade de medida para resistências

P Potência envolvida em um componente ou sistema físico

q Carga elétrica

R Variável de resistência de um resistor elétrico

ρ Variável generalizada de esforço ou de fluxo

v Tensão elétrica

V Volt - Unidade de medida para tensões elétricas

Z Impedância

Lista de Abreviaturas

API *Application Programming Interface*

CSS *Cascading Style Sheets*

DFS (Grafos) *Depth-First Search Algorithm*

DFS (Representação Sistemas Físicos) *Diagrama de Fluxo de Sinais*

DOM *Document Object Model*

HTML *HyperText Markup Language*

JSON *JavaScript Object Notation*

POO *Programação Orientada à Objetos*

SPA *Single-Page Application*

SVG *Scalable Vector Graphics*

UI *User Interface*

UML *Unified Modeling Language*

XML *eXtensible Markup Language*

Sumário

1 Introdução **16**

1.1 Justificativa 18

1.2 Objetivos 18

1.2.1 Objetivos Gerais 18

1.2.2 Objetivos Específicos 18

1.3 Organização do Trabalho 19

2 Teoria dos Grafos **20**

2.1 Conceito de Grafo 20

2.2 Definições Gerais 21

2.3 Classificações de Grafos 21

2.3.1 Grafos Direcionados 22

2.3.2 Grafos Conexos 22

2.3.3 Árvore 22

2.3.4 Árvore geradora de um grafo 24

3 Conceitos da Engenharia de Software **25**

3.1 Desenvolvimento Web 25

3.1.1 *Layout* e estilização 26

3.1.2 A linguagem Javascript 27

3.1.3 Canvas HTML5 27

3.2 Paradigmas da Programação 29

3.2.1 Programação Estruturada 29

3.2.2 Programação Orientada à Objetos 30

3.3 Estrutura de Dados 33

3.3.1 Estruturas Lineares 34

3.3.2 Estruturas não Lineares 34

4 Modelagem de Sistemas Físicos **37**

4.1 Variáveis do sistema 37

4.2 Propriedades dos componentes físicos 38

4.2.1 Representação matemática dos componentes físicos . . . 39
4.2.2 Outros componentes físicos . . . 44
4.3 Formas de representação de sistemas físicos . . . 44
4.3.1 Grafo de Ligação . . . 45
4.3.2 Diagrama de Fluxo de Sinais . . . 46
4.4 Regra de Mason . . . 47

5 Metodologia **49**
5.1 Tela Inicial . . . 50
5.1.1 Estrutura dos componentes . . . 51
5.1.2 Conectores e juntas . . . 54
5.1.3 Conexões . . . 57
5.1.4 Controle de Colisão . . . 62
5.1.5 Interface Gráfica . . . 66
5.1.6 Validação do sistema físico . . . 74
5.2 Painel de Resultados Parciais . . . 76
5.2.1 Interface Gráfica . . . 76
5.2.2 Construção do Grafo de Ligação . . . 77
5.2.3 Cálculo das matrizes de interconectividade . . . 79
5.2.4 Construção do DFS . . . 84
5.3 Apresentação da Função de Transferência . . . 85
5.3.1 Aplicação da Regra de Mason . . . 85

6 Resultados **89**
6.1 Organização do diagrama do protótipo . . . 89
6.1.1 Cenário 1 . . . 90
6.1.2 Cenário 2 . . . 90
6.1.3 Cenário 3 . . . 91
6.1.4 Cenário 4 . . . 92
6.2 Tela de Resultados . . . 93
6.3 Análise da Função de Transferência . . . 94
6.3.1 Cenário 1 . . . 95
6.3.2 Cenário 2 . . . 97
6.3.3 Cenário 3 . . . 98
6.4 Sistemas sem solução . . . 101
6.5 Discussão . . . 102

Conclusão **104**

Bibliografia **106**

Capítulo 1

Introdução

Sistemas de controle estão presentes de forma abundante em nosso meio, onde podem-se observar diversos fenômenos, sejam eles naturais ou criados pelo homem, para atuar em tarefas que de alguma forma desempenha um papel importante em nosso dia a dia. Desde foguetes, aviões ou outros meios de transporte ao sistema controle de açúcar presente no sangue de um corpo humano pelo pâncreas [Nise, 2013], mecanismos de controle são necessários em diversos aspectos e nos evidencia que a natureza já é composta por muitos desses sistemas que utilizam os conceitos supracitados. O papel do engenheiro é projetar um produto de acordo com o que ele observa ao seu redor a fim de que ele seja útil para tornar uma tarefa, inicialmente complicada, em algo simples de ser resolvido [Brown, 2001].

Além disso, um profissional que trabalha com controle precisa conhecer a fundo as propriedades dos componentes e ferramentas que vai utilizar no decorrer do projeto, bem como, suas características físicas e matemáticas [Maitelli and SILVA, 2005] [Franklin et al., 2009]. Possuir esse conhecimento vai ajudá-lo saber quais são as finalidades e limitações de cada elemento, proporcionando alcançar o resultado desejado com excelência e restando apenas a dificuldade de lidar com possíveis cálculos extensos e complexos, desafiando as suas habilidades relativas à teorias da álgebra e do cálculo, já que alguns componentes possuem características físicas de difícil compreensão, mas com papel de grande importância no contexto geral da solução.

Desde então, as teorias acerca deste assunto passaram a ganhar um enorme espaço dentre os temas estudados pela civilização, e, à medida em que a tecnologia se torna mais evidente, se torna possível a criação de sistemas mais sofisticados, com muitas entradas e muitas saídas. Portanto, a quantidade de equações requeridas para conseguir projetar uma simulação também aumenta [Ogata, 2010], tornando o trabalho do projetista uma tarefa cada vez mais difícil. Fica evidente, então a necessidade do auxílio computacional, que com a disponibilidade de microprocessadores mais potentes capazes de realizar um grande número de cálculos complexos em um curto espaço de tempo [STEMMER, 2001], possam identificar modelos mais sofisticados [Franklin et al., 2009] e projetar soluções

para problemas, até então, não resolvidos.

O engenheiro de controle pode contar com o processamento computacional, desde que esteja amparado por um software que consiga implementar os preceitos envolvidos na modelagem, apresentando resultados que simularão o funcionamento do projeto antes de qualquer tentativa real, evitando gastos com componentes que podem não ser utilizados. Surge assim um novo desafio: a implementação de um simulador capaz de modelar sistemas físicos com precisão. É claro que o software está cada vez mais presente no meio humano [Fernandes, 2003], e muitos de nós não mais nos imaginamos uma realidade em que passamos um dia sem interagir com nenhum tipo de sistema computacional [Pressman and Maxim, 2016]. No entanto, para implementar um programa computacional, é preciso que o desenvolvedor também compreenda todo o domínio ao qual será direcionada a solução [Rezende, 2006]. Definir a tecnologia (linguagem de programação), estruturar o código, organizar o diretório de pastas, implementar algoritmos para simplificar e resolver expressões polinomiais, preparar uma interface para que o usuário consiga utilizar o sistema. Todos esses são desafios enfrentados por um engenheiro de *software*. [Burd et al., 1999]

Mesmo dentro da computação, a evolução acontece em um ritmo acelerado. Aplicações de *software* que antes eram desenvolvidas para rodar em uma plataforma específica perdem espaço para novas soluções que podem rodar em um ambiente comum a diversos dispositivos. Basta apenas ter um navegador Web [Souza, 2016]. Esse cenário se agrava mais ainda quando fala-se a respeito de dispositivos diferentes do computador que também possuem uma unidade central de processamento. Além disso, novas linguagens de programação, com conceitos mais modernos e ferramentas de comunicação com o usuário mais atualizadas vão surgindo com esse mesmo paradigma de alta disponibilidade de *software*. Todas essas variáveis abrem uma margem para se pensar em uma solução de alta disponibilidade, com um bom nível de interação com o usuário capaz de resolver o problema de modelagem de sistemas de controle citado no início deste tópico.

Neste trabalho, focou-se na modelagem de sistemas físicos lineares utilizando a Regra de Mason. Para isso, foi levado em consideração um sistema já existente, implementado em 2005. O ModSym [Silva, 2005]. Nele o usuário consegue modelar sistemas físicos de natureza elétrica, mecânica (translacional e rotacional) e fluídica. O sistema foi desenvolvido na linguagem Delphi, disponível para Windows e Linux. Na aplicação é possível desenhar o sistema físico desejado nas suas diversas formas de representação (desenho gráfico, grafo de ligação e diagrama de fluxo de sinais) [Maitelli and SILVA, 2004] para se chegar a uma função de transferência que represente o comportamento nas variáveis de entrada e saída selecionadas. Existem outros sistemas que realizam tarefa similar, mas não abordam outras formas de representação do sistema [Silva, 2005].

1.1 Justificativa

Todos esses problemas apresentados se classificam como motivação e desafio ao mesmo tempo quando se fala no quesito de mesclagem de duas ciências diferentes (Sistemas de Controle e Programação de *Software*) para construção de um produto que representaria um avanço significativo para ambas as áreas, visto que existem poucos trabalhos que abordam acerca do tema. O ModSym surge como uma alternativa bastante inovadora, podendo ser utilizado inclusive para fins educacionais. Como foi desenvolvido em 2005, período em que o desenvolvimento Web ainda não era tão difundido, algumas abordagens podem vir à tona se forem utilizadas tecnologias mais recentes, se mostrando uma oportunidade perfeita para levar a mesma ideia do projeto para a Web, consumindo uma quantia bem inferior de recursos computacionais se comparado à sistemas dedicados para simulação. Focar o projeto na modelagem de sistemas físicos lineares vai permitir que a quantidade de recurso consumido seja reduzida, assim como acontece no Modsym.

Além disso, ferramentas como o Canvas do HTML5 podem oferecer uma série de novos recursos para se alcançar uma melhor comunicação com o usuário, apresentando elementos que ele já está acostumado a visualizar em outras páginas da Internet.

1.2 Objetivos

1.2.1 Objetivos Gerais

Implementar o AutoTF, um sistema Web capaz de modelar sistemas físicos lineares, apresentando informações de como o procedimento foi executado até chegar no resultado final: a função de transferência. O sistema deve conter interface gráfica para exibir o grafo de ligação do sistema, as matrizes parciais e final de interconectividade entre os componentes, o diagrama de fluxo de sinais e, por fim, a função de transferência, assim como mostrar também, informações, em cada estágio, quando existir algum impedimento para a execução das próximas etapas.

1.2.2 Objetivos Específicos

- Utilizar a linguagem Javascript para implementar o sistema, tornando o AutoTF um sistema acessível a partir de um navegador Web, sem dependência de sistemas operacionais;

- Apresentar um projeto interativo, de modo que o usuário tenha uma boa experiência de uso do sistema para modelar seus projetos, sem ficar perdido nas etapas intermediárias até a função de transferência;

- Apresentar interfaces de apresentação dos resultados de forma mais detalhada possível para justificar os resultados posteriores obtidos pelo AutoTF, reduzindo assim, a possibilidade de falhas;
- Obter funções de transferência em sua forma mais simplificada possível para facilitar a verificação dos resultados;
- Comparar resultados obtidos com modelos existentes na literatura de Sistemas de Controle, mostrando os cálculos necessários para chegar aos resultados manualmente.

1.3 Organização do Trabalho

O restante do trabalho foi organizado de modo que o leitor possa inicialmente ter uma abordagem teórica a respeito do tema tratado, e recursos que foram utilizados no decorrer do projeto. Em seguida são apresentados os desafios e métodos que foram utilizados na construção do código, para que por fim, a exibição dos resultados obtidos com a solução desenvolvida mostrassem que o objetivo final do trabalho foi alcançado. O capítulo 2 faz uma introdução à teoria dos grafos, recurso que foi bastante utilizado na construção deste trabalho. O capítulo 3 aborda tópicos da engenharia de *Software*, onde foram utilizadas diversas técnicas para manter uma boa organização na construção do AutoTF, que em sua versão inicial já possui um extenso código fonte. O capítulo 4 é o que aborda conceitos de sistemas de controle e modelagem de sistemas, tópico principal presente nos objetivos. O capítulo 5 relata o processo de implementação desde as dificuldades encontradas no desenvolvimento da interface gráfica, na construção do desenho do sistema no *Canvas*. Por fim, o capítulo 6 apresenta os resultados obtidos, discutindo a respeito do cumprimento dos objetivos propostos nas seções anteriores.

Capítulo 2

Teoria dos Grafos

Para a implementação de um projeto como o AutoTF é necessário destacar algumas informações a respeito de grafos. Os conceitos de modelagens de sistemas físicos estão diretamente voltadas ao uso dessa ferramenta, que possibilita a criação de modelos que representem situações mais próximas do mundo real. Este capítulo vai nos trazer o conceito principal de um grafo, bem como suas diversas classificações e principais características.

2.1 Conceito de Grafo

Grafos são modelos capazes de representar situações, problemas ou qualquer cenário característico de conexões entre elementos. Se considerarmos uma definição matemática, podemos definir grafo como um par de conjuntos (V, A), sendo V o conjunto dos vértices e A o conjunto das arestas. [Feofiloff et al., 2011].

Enquanto um vértice representa uma entidade que pode ser relacionada de alguma forma com uma outra entidade de mesma origem, uma aresta é denotada como um par de vértices v,w indicando que existe uma conexão entre eles. Os vértices v e w são então denominados vértices adjacentes [Costa, 2011]. Não é possível que um grafo possua a mesma aresta mais de uma vez no conjunto A, ou seja, arestas com mesma direção (caso o grafo seja direcionado) paralelas.

Graficamente falando, os vértices são representados como pontos ou circunferências com uma identificação ao centro. As arestas são desenhadas como retas que ligam os vértices uns aos outros, podendo ter ou não uma seta indicando sua direção. A ausência da seta indica que a conexão ocorre para ambos os lados. As arestas poderiam ter também um valor que definiria o seu peso no grafo.

A figura 2.1 mostra um modelo simples de grafo com vértices A, B, C e arestas (A,B),(A,C),(B,C) com pesos $[2, 4, 5]$, respectivamente.

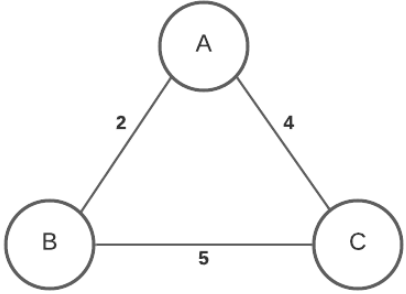


Figura 2.1: Exemplo de um grafo simples

2.2 Definições Gerais

Para efetuar uma análise coesa em uma estrutura de grafo, é necessário entender a nomenclatura e definição de suas características. Os conceitos de subgrafo, caminho, ciclo e grau [Mota, 2019] estão ligados diretamente à implementação de algoritmos de busca e manipulação de grafos, e devem ser devidamente estabelecidos para que situações mais complexas possam ser alcançadas.

- **Passeio:** Uma sequência de vertices v_1-v_2 existentes em um grafo G, de modo que v_1 é adjacente a v_2 e componente de origem de uma aresta que tem como destino o vértice v_2.
- **Caminho:** É um passeio no grafo que não apresenta repetição de vértices
- **Ciclo:** É um passeio existente no grafo, de modo que o seu vértice inicial e o vértice final sejam idênticos. Na figura 2.1, o caminho A,B,C,A forma um ciclo
- **Grau de um vértice:** Numero de arestas conectadas à um nó, sejam elas de entrada ou de saída, isto é, arestas associadas ao vértice analisado como ponto origem ou destino
- **Subgrafo:** É um grafo H formado à partir de um grafo G, cujo conjunto de vértices e arestas de H está contido no conjunto de G. Neste caso H é denominado o subgrafo, enquanto G é o supergrafo.

2.3 Classificações de Grafos

A grande quantidade de atributos que podem ser definidos em um grafo faz com que cada configuração defina uma possível classificação para todas as representações baseadas nela. Assim, serão apresentadas alguns possíveis tipos de grafos existentes, definindo assim novos conceitos à partir de cada um deles.

2.3.1 Grafos Direcionados

Também conhecidos como dígrafos [Prestes, 2016] ou grafos dirigidos, são grafos cujas arestas apresentam um sentido de ligação entre os vértices. Enquanto grafos não direcionados são compostos apenas por arestas antiparalelas, isto é com mesmos vértices, porém sentidos opostos, os grafos direcionados são compostos por arestas singulares. Conseguir traçar um caminho do vértice A para o vértice B, não significa que à partir do B será possível acessar mais uma vez o A. A figura 2.2 apresenta alguns exemplos de grafos dirigidos.

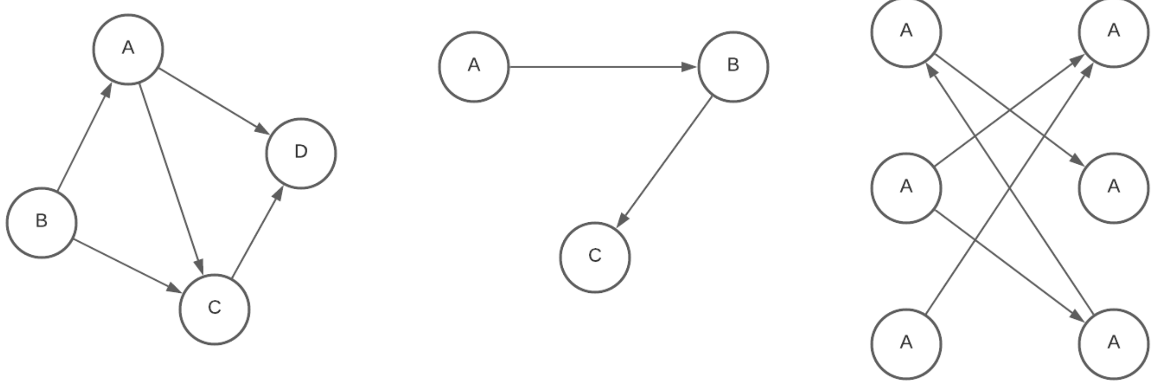


Figura 2.2: Exemplos de grafos dirigidos

2.3.2 Grafos Conexos

Os grafos também podem ser classificados quanto à sua conectividade, isto é, se é possível encontrar um caminho entre qualquer par de vértices que ele possui [Lucchesi, 1979]. Quando algum grupo vértices se encontra "isolado"de um outro grupo, o grafo é dito desconexo, e cada um desses grupos é chamado de elemento conexo do grafo. A estrutura mostrada na figura 2.2, se considerada como um único grafo, pode ser denominada como grafo desconexo, com três componentes conexas. Caso cada grafo seja analisado individualmente, eles serão classificados como grafos conexos.

Quando cada vértice de um grafo é adjacente a todos os outros vértices pertencentes ao conjunto v, esse grafo é dito totalmente conexo. Por outro lado, quando todos os vértices não possuem arestas que os interligam a um outro, o grafo é chamado de totalmente desconexo [Prestes, 2016].

2.3.3 Árvore

De modo geral, uma árvore é um grafo conexo que não apresenta nenhum ciclo em sua estrutura [Soares de Melo et al., 2014]. Normalmente são utilizadas para representar

cenários de hierarquia em algum ambiente ou contexto cujo problema deve ser resolvido. Como esse tipo de estrutura possui características bem específicas. À partir de sua definição, deve existir um vértice que ficará no topo de todos os outros, denominado de raiz da árvore [Prestes, 2016]. A figura 2.3 mostra alguns exemplos de estruturas organizadas com essas definições.

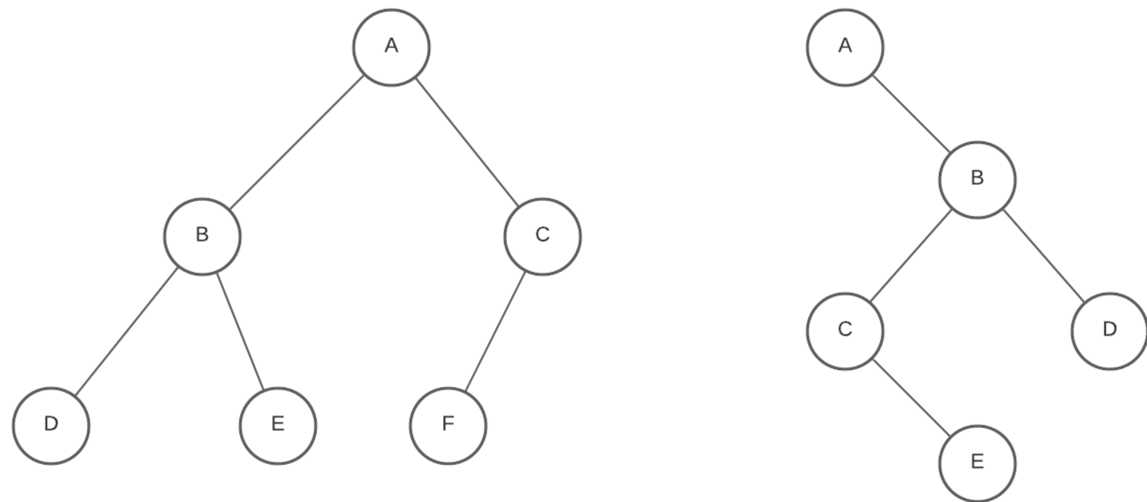


Figura 2.3: Exemplos de árvores

Assim como a raiz, algumas outras definições podem ser estabelecidas para caracterizar os diversos tipos de árvore existentes. Como visto na figura anterior, a estrutura é normalmente representada de cima para baixo. Os nós que estão abaixo, dependentes dos que estão acima para que sejam conectados com a raiz, são chamados de nós filhos, enquanto o que está acima é chamado de nó pai. A característica que define a quantidade de filhos que um dado vértice possui é também chamado de grau. Para cada nó existe também o nível, que representa a distância entre ele e o nó raiz. A maior distância encontrada em entre um determinado nó e a raiz de uma árvore é definida altura da árvore, que, por definição, estará sempre envolvendo, além da raiz um nó que não possui filhos. Todos os vértices que não possuem dependentes, ou seja, grau zero, são chamados de folhas.

Nesse contexto, surge então a definição de árvore binária, que tratam-se de estruturas cujo vértices possuem grau menor ou igual a 2, isto é, podem ter no máximo dois filhos. Para esse tipo de grafo, é possível aplicar uma nomenclatura a cada um dos nós herdeiros, sendo um deles o filho da direita e o outro, o filho da esquerda. Com base nessa característica, é possível abordar problemas mais específicos, simplificando o uso de rotinas de busca de dados ao considerar árvores binárias. As estruturas apresentadas na figura 2.3 são árvores binárias.

Por ser um grafo, todas as características já definidas neste capítulo para esses elementos são aplicáveis também à árvores. Dessa forma, pode-se dizer que toda a estrutura

abaixo de um vértice qualquer de uma árvore forma, juntamente à ele, uma subárvore.

Ao analisar uma estrutura quanto sua conectividade, é possível que todos os componentes conexos sejam árvores. Para este caso específico, a estrutura como um todo pode ser denominada floresta. Considerando, por exemplo, a estrutura da figura 2.3 como um único grafo, e possível defini-la como floresta.

2.3.4 Árvore geradora de um grafo

Dado uma estrutura em árvore T, esta será chamada de árvore geradora de G caso ela seja um subgrafo de G [Viana, 2016]. Por ser uma árvore, T não possui ciclos em sua estrutura, trazendo consigo aplicações interessantes para identificar ciclos em G. A figura 2.4 apresenta um exemplo para um grafo com sua árvore geradora.

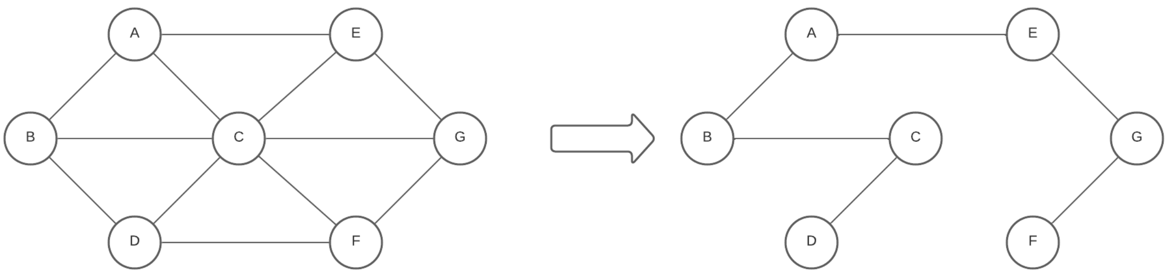


Figura 2.4: Exemplo de grafo e sua árvore geradora

Os algoritmos para obter árvores geradoras à partir de grafos normalmente seguem um mesmo procedimento para completar a sua função [Carvalho and Itália, 2005]. Em um primeiro momento é verificado se o grafo avaliado possui ciclos. A não ocorrência de ciclos indica que o próprio grafo é sua árvore geradora. Caso existam ciclos, a estratégia adotada é a remoção (ou adição) das arestas uma a uma, afim de manter as condições de conectividade acíclica, de acordo com a definição principal de árvore. O grafo resultante após esse processo é a árvore geradora.

A partir do processo de obtenção de árvores geradoras, é fácil perceber que cada uma das arestas removidas corresponde a um ciclo do grafo original. Isso pode ser utilizado para analisar individualmente os laços de uma estrutura, sabendo que o primeiro vértice do ciclo formado após adição da aresta removida na árvore geradora é formado pelo nó de origem dessa aresta.

Quando o grafo original é composto por arestas valoradas, surge o conceito de árvore geradora mínima, que atribui uma restrição ao processo de definição das árvores geradoras. A soma dos valores de todas as arestas da árvore resultante deverá ser a menor possível para que ela seja uma árvore geradora mínima do grafo [Viana, 2016].

Capítulo 3

Conceitos da Engenharia de Software

A solução presente neste trabalho exige diversos conhecimentos presentes na engenharia de *software* para que sua implementação fosse realizada, e assim prover a reutilização de código, manter a boa organização de um código fonte extenso tornando possível sua representação através de uma documentação. Além disso, a modalidade Web requer algumas práticas que diferenciam da forma de desenvolvimento para sistemas de outras modalidades.

3.1 Desenvolvimento Web

O desenvolvimento Web é uma sub-área da tecnologia voltada para a construção de aplicações que envolvam a Internet para o seu funcionamento. Não se restringe à sites ou páginas que podem ser acessadas através de um navegador, mas sim qualquer recurso que se precise conectar a um servidor para prover suas funcionalidades, como aplicativos *mobile*, banco de dados ou simples rotinas que realizem algum tipo de automação ao serem executadas [Miletto and de Castro Bertagnolli, 2014]. Esse conjunto de aplicações definem a Internet como é conhecida hoje, criando propósitos para seu uso.

Por esse motivo, as tecnologias envolvendo programação Web não param de evoluir fazendo com que as dificuldades e desafios se tornem também mais numerosos [Loudon, 2018]. Problemas como disponibilidade, gerenciamento de um grande volume de dados, autonomia e longevidade da aplicação se tornam verdadeiras dores de cabeça para desenvolvedores, que precisam se atualizar a todo instante, acompanhando as mais recentes novidades. Seja um novo *framework*, uma linguagem de programação mais adequada a um cenário específico ou até mesmo um novo padrão de escrita de código.

De forma geral existem duas modalidades distintas que os profissionais de desenvolvimento web podem atuar: O *front-end* e o *back-end*. Há ainda aqueles que têm domínio em ambas as áreas, denominado *fullstack*. Para cada uma dessas áreas, é necessário que sejam estudados conhecimentos específicos, de modo que elas se completem na construção do projeto como um todo [Pérez Ibarra et al., 2021]. Esse método de organização é o que

define por exemplo, qual máquina que executará o processamento de dados solicitados pelo usuário. É conveniente destinar essa tarefa para o servidor *back-end*, que normalmente é caracterizado por uma máquina com características mais afinadas à propósitos dessa natureza. Por outro lado, a disponibilidade de uma linguagem de programação altamente capaz de gerenciar os recursos da máquina do cliente e uma boa implementação do projeto possibilitam que processamentos mais complexos sejam feitos no *front-end*, permitindo uma maior agilidade no tempo de resposta da aplicação e aumentando a capacidade visual e interativa da interface gráfica. Dessa forma, podemos dizer que o *front-end* é o responsável pela comunicação com o usuário, seja através de páginas web ou de aplicativos, enquanto o *back-end* atua nos serviços que são executados em segundo plano, provendo recursos para o funcionamento correto da aplicação [Pressman and Maxim, 2016].

Na construção deste trabalho, a necessidade do uso de técnicas de *front-end* foi predominante, uma vez em que foi atribuída ao Javascript toda a função de processamento de dados. Afinal, o sistema foi construído com o propósito de ser alocado facilmente à páginas Web. Além disso, técnicas utilizadas para *layout* e organização dos elementos da tela representaram um fator de grande importância no desenvolvimento da solução.

3.1.1 *Layout* e estilização

A interface das páginas Web como conhecemos hoje surgiu à partir do conceito de hipertexto, oriundo das ideias de Tim Benners-Lee [Miletto and de Castro Bertagnolli, 2014] que consiste em uma nova forma de organização dos elementos através de *links* capazes de direcionar o usuário para uma área específica da página ou até mesmo para outras telas. Na prática, isso é possível graças à linguagem de marcação utilizada na construção de elementos que são interpretados pelo navegador, o HTML. A partir desse conceito, novas abordagens vão surgindo para melhorar as características visuais da Internet, e consequentemente, novos elementos são criados, proporcionando uma melhor experiência ao usuário [Zemel, 2015]. Nesse contexto, uma linguagem de estilo é necessária para prover ferramentas ao desenvolvedor modificar a identidade visual dos elementos HTML, abrindo uma série de possibilidades para novas ideias.

O CSS é uma linguagem que tem como propósito definir propriedades dos elementos que atuem na sua forma de apresentação e posicionamento na janela [Mansfield, 2005]. Também é a partir dela que são desenvolvidos os recursos da chamada "responsividade", característica que define a capacidade de adaptação de páginas web para exibição em telas pequenas, fazendo com que os elementos se reorganizem a fim de manter a página funcional. No entanto, elaborar todo o planejamento visual de uma página web pode ser uma tarefa que consome bastante tempo, e, à depender da urgência e dos objetivos do desenvolvedor, pode se tornar um impedimento no cumprimento da entrega do projeto em tempos reduzidos.

Para atender a este propósito, alguns *frameworks* que trazem padrões de estilos pré definidos foram implementados. O *Bootstrap* é uma dessas ferramentas. Foi lançado em agosto de 2011 por Mark Otto e Jacob Thornton enquanto ainda trabalhavam no Twitter [Spurlock, 2013]. Os *layouts* padronizados fornecidos pela ferramenta são comumente encontrados em temas de diversas páginas de sistemas Web disponíveis hoje em dia, principalmente em sistemas de *dashboards* que conta com inúmeros componentes gráficos para representar estatísticas de um certo conjunto de dados para o usuário. O *Bootstrap* também trás uma série de recursos para a implementação de responsividade dos componentes, atribuindo por exemplo, diferentes tamanhos à uma caixa de texto, de acordo com o tamanho total da tela onde a página está sendo exibida.

3.1.2 A linguagem Javascript

Javascript é a linguagem de programação Web. É com ela que é possível implementar os efeitos das páginas, máscaras nas caixas de texto, tratamentos de eventos e muitos outros recursos dinâmicos. Utilizando apenas HTML e CSS, o melhor resultado que pode ser obtido é uma página bem decorada, responsiva, mas com dados estáticos [Silva, 2010]. Se ela tiver recursos para atualizar automaticamente, ou buscar dados, ou qualquer tarefa mais complexa, é certo que o Javascript de alguma forma está envolvido nesse processo, se apresentando como o elemento que faz a "mágica"acontecer, provendo assim ao usuário um bom nível de usabilidade ao navegar na Internet.

Apesar de ter as raízes à partir do Java e da linguagem *Script*, o padrão do Javascript já se difere muito das suas origens, tendo deixado vários conceitos adotados por essas linguagens para trás. A única semelhança que ainda possui com o Java é a sua sintaxe [Flanagan, 2004], facilitando bastante o aprendizado para aqueles que já possuem uma certa familiaridade com programação.

Browsers estão ficando cada vez mais sofisticados [Mowery and Shacham, 2012] trazendo mais poder à linguagem Javascript que surge como uma ferramenta poderosa capaz de realizar processamentos em alta velocidade tornando possível realizar atualizações em milésimos de segundo e fazendo com que o usuário nem perceba que um determinado elemento ou parte de uma página foi recarregada. Fato este que abre possibilidades para desenvolvimento de novos recursos, como acontece de forma geral na tecnologia.

3.1.3 Canvas HTML5

O *Canvas* HTML5 é uma ferramenta para desenho dentro do contexto de uma página Web. Surgiu com o lançamento da versão 5 da linguagem de marcação HTML, possibilitando, através de sua API Javascript [Fulton and Fulton, 2013], manipular o espaço reservado desenhando linhas, formas circulares, retângulos, inserindo imagens e capturando eventos à partir dos periféricos de entrada (como teclado e mouse) para executar

diferentes comportamentos de acordo com o ambiente desenvolvido. Recursos desse tipo fazem com que o sistema apresente uma boa aparência e seja intuitivo para o usuário, além de serem fundamentais para implementação de elementos dinâmicos, como jogos e animações, por exemplo.

O *Canvas* se baseia nas coordenadas cartesianas para definir o posicionamento de seus objetos, de modo que o eixo das ordenadas encontra-se invertido. Áreas superiores da tela possuem menor valor em relação à áreas inferiores, conforme mostra a figura 3.1. Dessa forma, quanto maior for o valor da coordenada y, mais baixo estará o objeto.

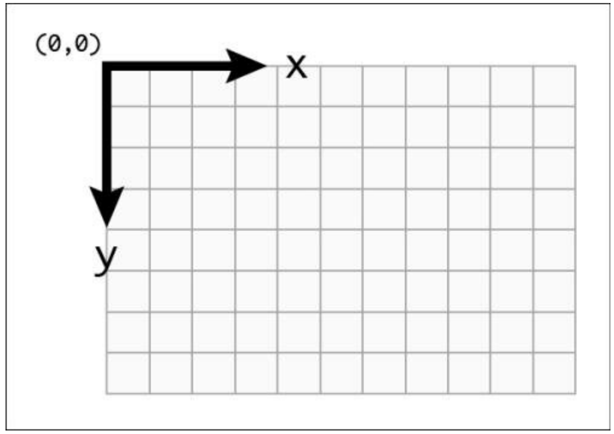


Figura 3.1: Sistemas de coordenadas no Canvas HTML5

Apesar de se tratar de uma tecnologia poderosa, ainda existem poucas aplicações Web que utilizam a ferramenta para construir áreas interativas em suas páginas, por normalmente não necessitarem do recurso, visto que formulários e textos estáticos ainda são mais comuns. Ao precisar de efeitos gráficos na aplicação, é conveniente utilizar uma tecnologia alternativa mais antiga que apresenta uma forma de implementação bem mais simples, mas que atende também as necessidades do sistema.

Canvas x SVG

O SVG é uma linguagem de descrição de objetos 2D a partir de vetores gráficos, imagens ou texto [Ferraiolo et al., 2000], que permite criar elementos interativos e animados utilizando a linguagem XML. A existência de uma linguagem *script* que atua diretamente no DOM do SVG faz com que cada objeto interno da interface resultante possa ser acessado, associando à eventos, por exemplo, garantindo a compatibilidade necessária para o ambiente Web. Junto ao *Canvas* HTML, são as tecnologias mais utilizadas na Internet para implementação de elementos personalizados, efeitos e animações gráficas.

Dessa forma, ao começar um novo projeto que possui como recurso principal gráficos dinâmicos, o desenvolvedor precisa tomar uma decisão inicial que reflete na escolha de qual tecnologia será utilizada. O *Canvas* é uma tecnologia de mais baixo nível, pois opera diretamente na construção do *bitmap* da área reservada por ele, enquanto o SVG

é de alto nível fazendo com que o programador apenas descreva o objeto em vetores que ele quer mostrar em tela. Dessa forma, podemos perceber que o *Canvas* trás uma maior flexibilidade e desempenho, além de evitar que o programador tenha que lidar com o XML, entretanto, o desenvolvedor deve estar preparado para escrever uma grande quantidade de código utilizando sua API [Eis and Ferreira, 2012].

Todos as interfaces que são resultantes do SVG também podem ser construídas no *Canvas*, [WPInfo, 2010] cujo limite é definido pela disposição do desenvolvedor em escrever grandes volumes de código, juntamente com uma boa lógica matemática para posicionar os elementos e cadastrar eventos. A combinação desses requisitos torna possível o uso de uma ferramenta poderosa podendo resultar em excelentes produtos.

Por outro lado, se a necessidade é fugir da complexidade em modelar objetos e formas personalizadas, talvez descrevê-los seja uma opção, visando economia de tempo e simplicidade de código. O SVG também se apresenta como uma excelente opção, que apesar de não ter a mesma performance oferecida pelo *Canvas*, consegue gerar ótimos resultados gráficos, garantindo a simplicidade de sua construção. De qualquer forma, a combinação das duas tecnologias também se apresenta como um possível cenário, visto que elementos simples, como ícones, normalmente escritos em SVG, são frequentemente necessários em quase todos os tipos de aplicação Web.

3.2 Paradigmas da Programação

A programação de *software* tem se tornado um processo cada vez mais importante para a humanidade, que se encontra em crescente sintonia com a tecnologia. Por isso, novas ferramentas vão surgindo todos os dias com objetivo de facilitar e simplificar a tarefa de desenvolvimento de *software*. Com o passar dos anos, novos métodos para programar vão ganhando espaço entre os desenvolvedores, que hoje contam com dois principais conceitos para elaborar novas soluções: a programação estruturada e a programação orientada à objetos que trazem diferentes abordagens acerca do assunto.

3.2.1 Programação Estruturada

De acordo com [Jackson, 1975], programação estruturada é uma forma de programação de computadores que preconiza que todos os programas possíveis podem ser reduzidos à apenas três estruturas: sequência, decisão e iteração(repetição). Foi o padrão dominante adotado antes do surgimento da programação orientada a objetos. Surgiu para evitar a práticas dos *GOTOs*, instruções que faziam com que o código não fosse sequencial, causando vários "pulos"durante sua execução. Dessa forma, a manutenção e depuração se tornavam tarefas exaustivas e improdutivas.

O padrão estruturado surgiu como meio para evitar essa prática, defendendo que as

rotinas de comando seriam o suficiente para tornar o código mais claro, e manter seu funcionamento sem utilizar comandos de mudança de linha, dando origem às instruções de decisão (*if* e *switch*) e de repetição (*for* e *while*).

3.2.2 Programação Orientada à Objetos

A programação orientada a objetos (POO) é um conceito que trás uma nova abordagem à maneira como são desenvolvidos os *softwares*. Seu propósito é fazer com que o código fonte implementado se assemelhe a objetos que encontramos no mundo real, facilitando a visualização do problema ou do ambiente o qual se quer construir [Farinelli, 2007]. Em torno disso, duas novas definições são necessárias para que todas as teorias acerca de POO sejam formuladas: As Classes e Objetos.

Uma classe é uma entidade que define todas as propriedades relacionadas à objetos comuns. É nela que são implementados os atributos e métodos, que correspondem ao conjunto de características que seus elementos vão possuir e ações que poderão ser executadas por eles [Ricarte, 2001]. Os objetos são instâncias dessas classes, que surgem como modelos gerados à partir de um molde, com valores atribuídos aos atributos da classe [Gonçalves, 2018]. Nesse sentido, se faz possível a criação de algumas técnicas que contribuem para solução de problemas encontrados na programação estruturada. Os conceitos de encapsulamento, herança, interface e polimorfismo é o que proporciona o diferencial da programação orientada a objetos, surgindo como solução inovadora para um mundo até então estruturado.

Encapsulamento

A implementação de uma classe com seus atributos e métodos proporciona muitas vantagens trazendo para mais perto do mundo real um conjunto de características e ações que o representam. No entanto, o mau uso dessas técnicas pode acarretar em sérios problemas na solução, ocasionando em resultados diferentes do esperado. Esses atributos de classe precisam então ser encapsulados. O encapsulamento é uma técnica da POO que permite atribuir permissões para edição de atributos por outras classes [OTSUKA and ZANELATO, 2012]. Métodos *getters* e *setters* são implementados para garantir que não haverão inconsistências no valor desses atributos, de modo que o *getter* é chamado quando se deseja ler o valor de uma propriedade, e o *setter* quando for preciso atribuir um valor à ela.

Herança

A herança é um conceito da orientação à objetos que permite criar uma nova classe com características de uma outra classe já existente. Essa nova classe vai "herdar"os atributos e métodos da classe mãe todos os seus objetos também serão considerados

objetos herdeiros [Farinelli, 2007]. Um exemplo clássico que podemos citar são as classes Animal, Cachorro e Gato. Todos os animais possuem peso, altura, cor do pelo e emitem um som. Logo, podem ser definidas diretamente na classe animal. Assim, ao tornar Cachorro e Gato classes herdeiras de Animal, já serão construídas, por padrão, com esses mesmos atributos.

Esse é um recurso muito utilizado para redução de código repetido e padronização das características de duas classes diferentes. Isso não impede que atributos ou métodos específicos sejam criados nas classes filhas representando características particulares do objeto gerado.

Interfaces

É possível então perceber que a herança é um recurso bastante útil em situações de similaridade entre classes, fazendo com que classes herdeiras possuam também atributos de quem herda. Mesmo assim, uma classe mãe pode ter seus próprios objetos que implementam apenas as características em comum com as classes específicas. Ao existir situações em que essa classe não atue ativamente no contexto do projeto, mas ainda sim seja necessária para definir o comportamento de quem a segue, surge a ideia de interfaces na POO.

Interfaces são entidades que descrevem uma série de parâmetros esperados de quem a implementa. Dessa forma não é necessário que exista nenhuma rotina em seu código [Stroustrup, 1988] [Ricarte, 2001] além da nomenclatura, o tipo de retorno e os parâmetros dos atributos e métodos. São bem úteis para definir padrões e obrigações de classes que a implementam. Com esse conceito é possível associar objetos à situações que ele deve saber como resolver. Como exemplo podemos citar um sistema escolar que para acessar seus recursos, é necessário que o usuário se autentique. As classes Diretor, Professor e Aluno, que são as entidades que possuem acesso ao sistema e devem implementar uma interface Autenticável, que contem um método *autenticar*. Assim, qualquer uma dessas classes obrigatoriamente terão que implementar, de acordo com cada caso, esse mesmo método.

Polimorfismo

Polimorfismo é o conceito da programação orientada à objetos que permite duas ou mais classes distintas executar métodos com mesma nomenclatura, com comportamentos diferentes [Farinelli, 2007], desde que pertençam à mesma superclasse. Esse recurso é bastante útil quando se deseja executar uma atividade genérica, independente de qual objeto esteja sendo referenciado. Podemos utilizar o exemplo que citamos na seção de herança ao implementar o método emitir som na classe Animal. As classes Cachorro e Gato terão suas próprias implementações deste método. Ao chamar a função de emitir som em cada

um dos objetos, a rotina executada será respectiva à cada classe da qual a instância se originou.

Diagramas de Classes UML

Ao implementar uma aplicação cuja estrutura utiliza os conceitos da orientação à objetos, um desafio comum do desenvolvedor é documentar à respeito de tudo que foi ou será implementado, proporcionando um certo nível de comunicação com outros desenvolvedores que precisarão corrigir eventuais erros ou implementar novos recursos na aplicação. Com esse propósito foi elaborado o diagrama de classes UML, que é uma ferramenta voltada para especificar os componentes da aplicação indicando a forma na qual interagem uns com os outros.

A UML é um padrão de diagramas que são utilizados para representar objetos ou comportamentos relacionados à um projeto computacional, trazendo diferentes visões para abordar tópicos específicos da solução ou produto desenvolvido [Rezende, 2006]. Dentre os todos os diagramas, o mais utilizado é o de classes, que serve para apresentar um esquema gráfico apresentando as classes existentes no sistema. É possível ver também quais atributos e métodos cada uma possui, bem como seus relacionamentos com outras classes. A figura 3.2 apresenta um modelo de diagrama de classe UML.

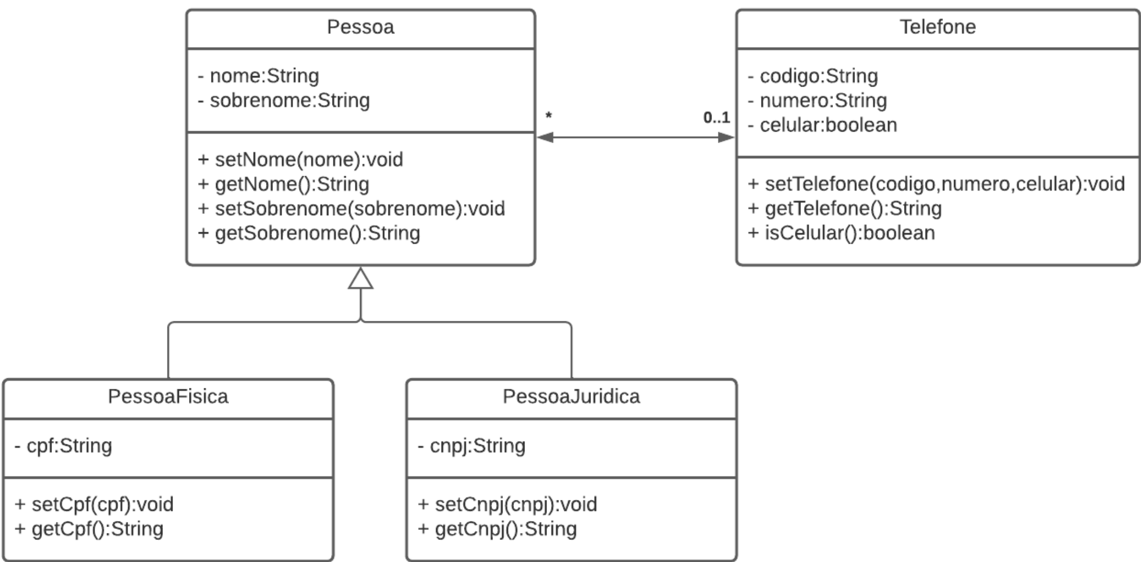


Figura 3.2: Exemplo de um diagrama de classes

Para elaborar ou interpretar um diagrama de classes é preciso entender seus componentes gráficos. Conhecendo as representações e símbolos existentes, pode-se saber qual mensagem a representação gráfica quer passar. Ao visualizar o exemplo da figura 3.2, conseguimos observar grandes entidades formadas por retângulos divididos em três partes. Essas são as classes, onde de cima para baixo estão dispostos o seu nome, os atributos e

os métodos que ela possui. Cada atributo ou método vem acompanhado de um símbolo à esquerda indicando quais classes podem acessá-lo (representação do encapsulamento), de modo que os principais símbolos são:

- **(+)**: Público - Acessível por qualquer outra classe
- **(-)**: Privado - Apenas a classe que implementa o atributo/método pode acessar
- **(#)**: Protegido - Apenas a classe que implementa e seus herdeiros podem acessar o recurso

As linhas que interligam as classes são chamadas de relacionamentos, e pode ter significados diferentes de acordo com o símbolo presente em suas extremidades.

- **Association** (Associação – conector sem pontas): interligam classes independentes, cuja existência não depende da outra, mas que estão associadas de alguma forma
- **Generalization** (Herança – conector com seta em uma das pontas): Especifica uma relação de herança entre duas classes. A seta ou triângulo vazado aponta para a classe mãe
- **Compose** (Composição – conector com um "diamante" pintado na ponta): Relacionamento que indica que a classe apontada depende da outra para que exista
- **Aggregate** (Agregação – conector com um "diamante" vazado na ponta): Indica que a classe apontada utiliza a outra para existir, podendo ser instanciada sem ela.

Dependendo da ocasião, ela poderá estar acompanhada de valores, próximos a cada entidade interligada. Esses valores são chamados de cardinalidades e indicam a característica do relacionamento em relação à quantidade de elementos envolvidos.

3.3 Estrutura de Dados

Na computação normalmente utilizamos dados de modo que eles estejam relacionados entre si, criando uma dependência para acesso dessas informações. Esses dados podem ser relacionados à uma mesma entidade, ou a um problema como um todo, tornando necessário diferentes formas de organização para tratar os diversos casos da maneira mais adequada. [BALIEIRO, 2015] Algumas topologias são mais indicadas para grandes volumes de dados, outras apresentam melhores resultados em operações de busca. A maneira como um novo elemento é inserido ou removido também pode impactar diretamente na resolução do problema. Dentre alguns dos tipos de estruturas de dados mais utilizados, podemos citar listas, pilhas e filas como tipos de estruturas lineares, e árvores e grafos representando as estruturas não lineares.

3.3.1 Estruturas Lineares

A estruturas lineares são caracterizadas por agrupar os dados e objetos de modo sequencial. Assim, para adicionar ou remover um novo elemento em uma posição intermediária, é necessário que todos os itens à frente sejam realocados [Ricarte, 2008]. As listas, também conhecidas como *Array*, são as estruturas mais simples utilizadas para manipular dados dessa maneira, e é a partir dela que surgem as outras estruturas lineares. Operações gerais como criar nova lista, inserir ou remover um elemento em uma determinada posição, verificar se está vazia, são pertencentes à este tipo de estrutura de dados, sendo reaproveitados nos conceitos das outras estruturas lineares.

A forma como cada elemento da lista referencia o objeto adjacente representa a modalidade da ligação, podendo acontecer basicamente de duas maneiras diferentes: estática ou dinâmica [Gudwin et al., 1998]. Uma ligação estática caracteriza-se por associar elementos adjacentes em posições vizinhas dentro do espaço de memória, de modo que para encontrar o próximo elemento da lista, basta incrementar o endereço de memória acessado.

Estruturas dinamicamente interligadas armazenam o endereço de memória de seus vizinhos, tornando possível que estejam fisicamente localizados em áreas distintas da memória. Com elas torna-se possível implementar variações como listas duplamente encadeadas que armazenam endereços do anterior e do próximo elemento, ou listas circulares, onde o último elemento guarda o endereço do primeiro. A figura 3.3 mostra um exemplo de uma lista duplamente encadeada exemplificando as conexões entre os dados.

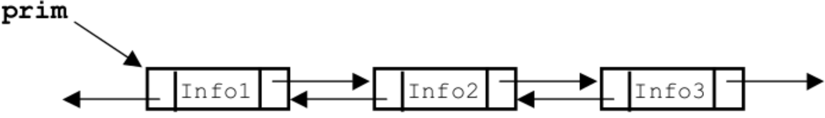


Figura 3.3: Lista duplamente encadeada

As pilhas e filas são estruturas implementadas à partir das listas encadeadas, e se diferenciam quanto a disponibilidade de acesso dos dados que as compõem. Enquanto nas filas o padrão adotado é o FIFO (*First In First Out*), obrigando que o primeiro elemento a entrar na estrutura seja o primeiro elemento a sair, nas pilhas utiliza-se o padrão LIFO (*Last In First Out*), onde o último elemento a entrar deve ser obrigatoriamente o primeiro a sair [BALIEIRO, 2015]. A figura 3.4 mostra essas estruturas.

3.3.2 Estruturas não Lineares

As estruturas não lineares são caracterizadas por possuir diversos tipos de objetos de tipos diferentes referenciados entre si, permitindo que os itens sejam adicionados ou removidos durante a execução do programa [Gudwin et al., 1998]. Elas também permitem a criação de topologias mais complexas, ideais para resolução de problemas de larga escala. As

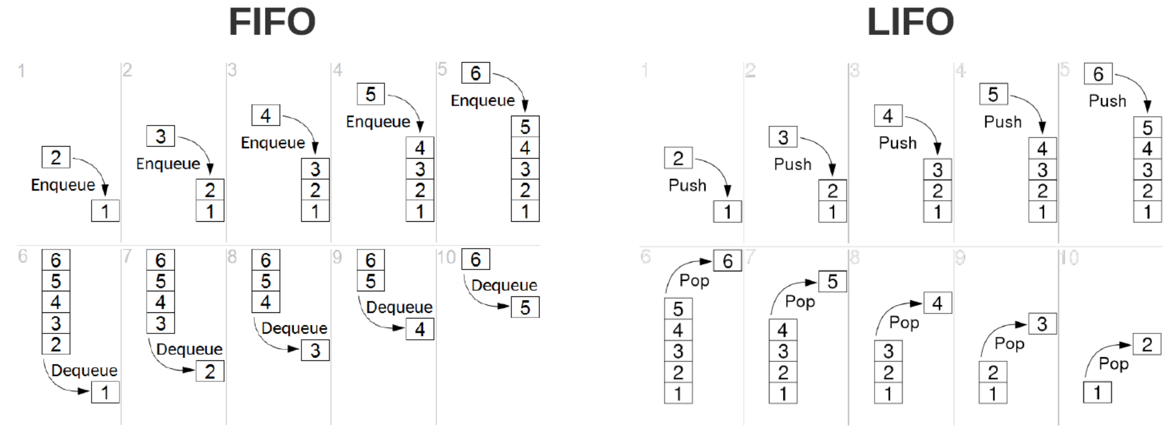


Figura 3.4: Estrutura de filas (FIFO) e pilhas (LIFO)

árvores e os grafos são os modelos mais conhecidos que implementam uma abordagem não linear.

Árvores

As árvores (figura 3.5) estão presentes em muitas situações que representam de alguma forma a ideia de hierarquias. Seja em uma organização de funcionários de uma empresa ou até mesmo no sistemas de arquivos de um sistema operacional, agrupando arquivos e diretórios, a árvore é uma estrutura essencial para simular esses eventos. Como explicado no capítulo 2, é composta por nós (ou vértices) e arestas, sendo representadas de cima para baixo. O nó é a unidade que contém os dados que serão processados, enquanto as arestas representa o elemento que liga os vértices. Podem ser unidirecionais ou bidirecionais, dependendo de como os nós podem ser acessados.

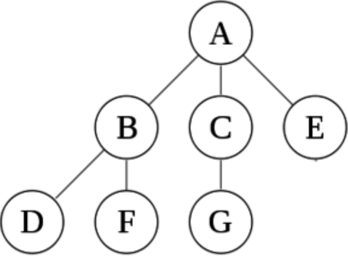


Figura 3.5: Estrutura de dados em árvore

Estruturas de árvore possuem uma restrição na conexão entre os vértices: os nós filhos podem possuir apenas um único nó pai. No entanto, é possível que um nó pai possua diversos nós filhos. Isso faz com que essa estrutura não possa apresentar qualquer tipo de ciclos em sua organização. Deve ser iniciada à partir de um vértice que é pai de todos os outros. Esse é o chamado vértice raiz. Sua implementação deve ser realizada

considerando essa restrição, principalmente no caso das árvores binárias, que permite apenas a existência de dois filhos por nó.

Grafos

Os grafos, em termos de programação, são estruturas semelhantes às árvores, compostos também por um conjunto de vértices e arestas. É correto dizer que toda árvore é um grafo, mas nem todo grafo é uma árvore. Nos grafos não existe restrição de conexão entre elementos como acontece com as árvores. Portanto, um mesmo vértice pode ser origem e destino de muitos nós, facilitando seu processo de representação em código, que apresenta algo semelhante a uma estrutura de redes, como mostrado na figura 3.6.

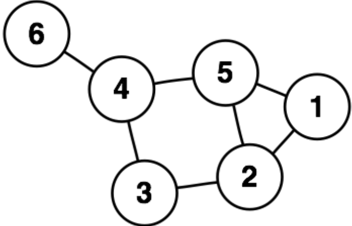


Figura 3.6: Estrutura de dados em grafo

Existem diversas formas de representação de um grafo dentro do contexto de programação de *software*. Dentre elas, podemos citar três que abrangem estratégias diferentes: Matriz de adjacências, lista de arestas e lista de adjacência [Wheatman and Xu, 2018]. Na matriz de adjacências, a estrutura de dados é representada por um *array* bidimensional em que as linhas representam os vértices de origem e as colunas os vértices destino. Portanto, para cada aresta existente no grafo, seu peso deve ser inserido na linha do nó de origem e na coluna do nó destino para que ele seja representado. Essa técnica é bastante útil para acessar um elemento específico, mas aloca muito espaço na memória, a depender da quantidade de vértices existentes.

A lista de adjacências consiste na representação a partir de uma estrutura chave-valor, onde cada chave representa um vértice do grafo, associado a listas de adjacências contendo referências para todos os nós conectados a esse vértice. É útil para preservar espaço em memória, visto que serão alocados recursos apenas para as arestas existentes. Também apresenta um bom desempenho em algoritmos de busca e acesso direto a elementos, considerando a dificuldade dos vértices possuírem uma lista de conexões extensa.

Por fim, a representação através de lista de arestas que consiste em uma lista de um par de valores, sendo o primeiro representando o nó de origem e o segundo o nó destino. É a forma de representação menos utilizada, porque apesar de implementar o conceito de economia de memória visto na lista de adjacências, seu desempenho em operações de busca deixa a desejar, sendo necessário que toda a lista seja percorrida para encontrar uma simples aresta, no pior caso.

Capítulo 4

Modelagem de Sistemas Físicos

A solução proposta por este trabalho é fundamentada em teorias de controle que são necessárias para o entendimento do comportamento de elementos físicos dentro do sistema. Neste capítulo, abordaremos conceitos essenciais para a modelagem de sistemas físicos, desde a formação das funções de transferência de cada componente, formas de representação variadas até a obtenção do modelo matemático que representa cada sistema à partir da aplicação da Regra de Mason. Esse foi o procedimento adotado pelo AutoTF para que seja possível cumprir os objetivos propostos neste documento. A versão definida para entrega deste trabalho adotou como escopo apenas elementos elétricos, os quais daremos mais destaque na construção dos textos abaixo, considerando também que os outros tipos de sistemas podem ser baseados nas mesmas fundamentações.

4.1 Variáveis do sistema

Um conceito básico para sistemas físicos pode ser definido como um conjunto de elementos físicos que ocupam uma porção do espaço-tempo, sujeitos a modificações temporais que atuam como manipuladores de energia através da interação com portas de entrada ou de saída [Wellstead, 1979]. Essas operações ocorrem por meio das variáveis do sistema que indicam de que forma e em qual direção ocorrem as transferências de energia.

Utilizando como base os conceitos físicos usados para sistemas elétricos, sabemos que a potência transferida para um componente ou um dispositivo elétrico equivale à:

$$P = vi \tag{4.1}$$

Onde "v"representa a diferença de potencial e "i"a corrente. A energia transferida para o sistema então se dá por:

$$E = \int_0^{t1} P\, dt = \int_0^{t1} vi\, dt \tag{4.2}$$

À partir das equações 4.1 e 4.2 podemos perceber que a transferência de energia ocorre em torno de duas variáveis [Maluf, 2020], uma que utiliza um ponto de referência para ser mensurado e outra que representa o fluxo de corrente que corre através dos fios. Essas são as características das variáveis de esforço e de fluxo, respectivamente. Elas são capazes de descrever o comportamento de cada sistema físico de modo que existe uma grandeza correspondente em cada natureza que atende essas características. A tabela 4.1 mostra as grandezas correspondentes às variáveis de esforço e fluxo para cada um dos diferentes cenários.

Tipo Sistema	**Esforço**	**Fluxo**
Elétrico	Tensão	Corrente
Mecânico	Velocidade linear	Força
Mecânico Rotacional	Velocidade angular	Torque
Fluídico	Pressão	Vazão
Térmico	Temperatura	Calor

Tabela 4.1: Variáveis de esforço e fluxo para cada tipo de sistema

Para cada uma das grandezas podemos perceber que para medir as que se categorizam como variável de esforço, é preciso adotar um ponto de referência. A velocidade, por exemplo, é uma grandeza que tem como base em seu conceito o ponto de origem no qual está sendo observado. A visão de um motorista ao ver ao seu lado um outro veículo ao seu lado com mesma velocidade instantânea, é de que ele está imóvel já que a diferença de velocidades entre eles é nula. Para calibrar um termômetro também é necessário aplicar um valor de referência para que ele possa aferir corretamente a temperatura de um ambiente ou um objeto, apresentando um valor resultante que faça sentido [Wellstead, 1979].

Por outro lado, as grandezas de fluxo se apresentam de modo que sempre estejam no mesmo sentido da transferência de energia aplicada. Quanto mais calor é transferido para um corpo, maior será sua energia. A mesma analogia pode ser utilizada para os outros tipos de sistema.

4.2 Propriedades dos componentes físicos

Como dito no início deste capítulo, cada uma das diferentes naturezas apresentadas na seção anterior possui componentes capazes de manipular o processo de transferência energética de diferentes formas. Os elementos básicos que podem ser encontrados são caracterizados de forma genérica a partir de três categorias. São elas:

- **Fontes de energia:** Elementos geradores de energia no sistema. Apenas uma grandeza é modificada, classificando os componentes em geradores de fluxo ou geradores de esforço.

- **Armazenadores de energia:** Elementos capazes de armazenar energia, classificados, assim como os geradores, em armazenadores de esforço ou armazenadores de fluxo.

- **Dissipadores de energia:** Componentes capazes de dissipar energia. Elemento único, não classificado em esforço ou fluxo como nos itens anteriores.

Esses componentes atuam no sistema utilizando apenas uma única porta de energia [Broenink, 1999]. A tabela 4.2 apresenta cada um dos componentes com as características mencionadas.

Natureza	**F. Esforço**	**F. Fluxo**	**A. Esforço**	**A. Fluxo**	**Dissipador**
Elétrico	F.Tensão	F. Corrente	Indutor	Capacitor	Resistor
Mecânico	Vel. linear	Força	Mola	Massa	Amortecedor
Mec. Rot.	Vel. angular	Torque	Mola	Inércia	Amortecedor
Fluídico	Pressão	Vazão	Tubulação	Reservatório	Atrito Flúido
Térmico	Temperatura	Calor	-	Capacitor T.	Resistor T.

Tabela 4.2: Componentes básicos de uma porta para cada tipo de sistema

4.2.1 Representação matemática dos componentes físicos

Cada um desses elementos apresentam equações matemáticas que os definem baseadas na quantidade de esforço e de fluxo que eles carregam, e portanto podem ser utilizadas para gerar uma equação que represente o comportamento do sistema. Essas equações são definidas de acordo com a característica do componente, e são genéricas para todos as naturezas citadas na tabela 4.2. Usaremos o modelo elétrico para fundamentar a formação da função que representa o ganho de energia equivalente de seus componentes.

Fontes de energia

As fontes de energia podem apresentar comportamentos variantes, como por exemplo as fontes elétricas de corrente alternada. Mesmo assim, elas não podem interagir com duas variáveis ao mesmo tempo [Wellstead, 1979], ou seja, as fontes variáveis devem ser compostas por funções que variam apenas no tempo. Uma fonte de esforço não pode ser representada por uma função cujo fluxo é uma das incógnitas, e vice e versa. De qualquer forma, para efeitos de simplificação, consideremos apenas fontes ideais, que apresentam valores constantes para a fundamentação da teoria proposta por esse trabalho. Sendo assim, fontes de fluxo fornecem valores constantes de fluxo ao sistema, da mesma forma que fontes de esforço contribuem com valores constantes da variável de esforço para o sistema. A figura 4.1 representa o comportamento das fontes de energia ideais.

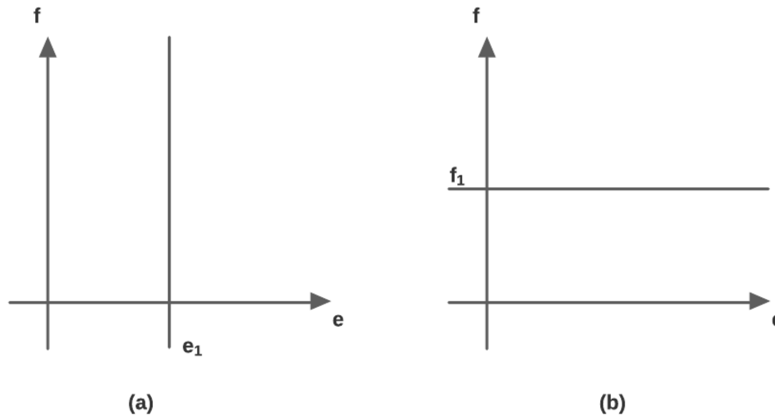


Figura 4.1: Comportamento das fontes de energia no sistema físico. a) Fonte de Esforço. b) Fonte de fluxo

Acumuladores de energia

Para os acumuladores e dissipadores, consideremos a equação 4.3 que representa o valor de energia envolvida à partir das variáveis de esforço e fluxo.

$$E = \int_0^t ef\, dt \tag{4.3}$$

Os acumuladores de energia são dispositivos capazes de armazenar energia à partir de uma das variáveis de sistema. Acumuladores de esforço são aqueles que armazenam energia através da retenção de esforço [Silva, 2005], de modo que a equação 4.4 representa o esforço acumulado no componente até um determinado tempo t.

$$e_a = \int_0^t e\, dt \tag{4.4}$$

Para acumular energia através de esforço, é preciso que é preciso que seja mantido um certo valor de fluxo passando através do acumulador. Dessa forma, o esforço acumulado pode ser representado por uma equação em função do fluxo, de modo que:

$$e_a = \phi(f) \tag{4.5}$$

Podemos assim reescrever a equação 4.3 para calcular a energia acumulada pelo componente.

$$E = \int_0^{e_a} f\, de_a = \int_0^{e_a} \phi(e_a)^{-1}\, de_a \tag{4.6}$$

Para o sistema elétrico, o componente que tem a função de acumulador de fluxo é o indutor [Broenink, 1999], que armazena tensão à partir de uma corrente induzida que gera um campo magnético, que por sua vez, resulta em um campo elétrico. Assim, à

partir da equação do indutor, sabemos que o valor de tensão entre seus terminais se dá pela equação 4.7

$$v = L\frac{di}{dt} \tag{4.7}$$

Dessa forma, é possível perceber que o esforço $e_a = v$ é representado em termos da variável de fluxo i. Logo, podemos definir o esforço armazenado e_a pela equação 4.8.

$$\begin{aligned} e_a &= \int_0^t v\, dt \\ e_a = \lambda &= Li = Lf \end{aligned} \tag{4.8}$$

Onde λ representa o esforço acumulado indicando o fluxo magnético concatenado do indutor, e L o valor de sua indutância. A figura 4.2 mostra o comportamento da função que representa a quantidade de esforço armazenado em um componente, considerando um armazenamento de fluxo linear.

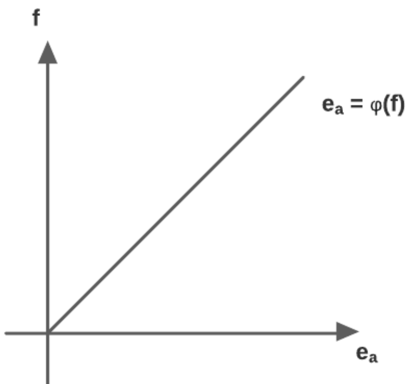


Figura 4.2: Relação entre esforço acumulado e fluxo em dispositivos acumuladores

De forma análoga, pode ser explicado o comportamento do acumulador de fluxo. São dispositivos capazes de armazenar energia através da retenção de fluxo [Wellstead, 1979]. A função 4.9 representa o fluxo acumulado no componente.

$$f_a = \int_0^t f\, dt \tag{4.9}$$

O armazenamento de fluxo requer que o armazenador esteja submetido à um esforço, de modo que o fluxo armazenado pode ser representado por uma função em termos do esforço, como mostra a equação 4.10.

$$f_a = \phi(e) \tag{4.10}$$

A partir da equação 4.3, chegamos à equação que calcula a quantidade de energia acumulada em um armazenador de fluxo:

$$E = \int_0^{f_a} e \, df_a = \int_0^{f_a} \phi(f_a)^{-1} \, df_a \tag{4.11}$$

No sistema elétrico, o capacitor é o elemento que representa um acumulador de fluxo [Broenink, 1999]. O seu comportamento é representado por:

$$i = C\frac{dv}{dt} \tag{4.12}$$

O fluxo acumulado no capacitor então é definido de acordo com a equação 4.13.

$$\begin{aligned} f_a &= \int_0^t i \, dt \\ f_a = q &= Cv \\ f_a &= Ce \end{aligned} \tag{4.13}$$

Onde q equivale à quantidade de carga elétrica presente no dispositivo, representando o fluxo acumulado e C a capacitância.

De forma genérica, o comportamento de armazenamento de fluxo pode ser descrito pelo gráfico da figura 4.3, considerando que o fluxo armazenado e o esforço apresentem relações lineares.

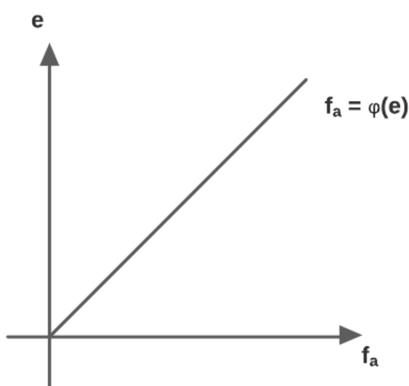


Figura 4.3: Relação entre fluxo acumulado e esforço em dispositivos acumuladores

Dissipadores de energia

Os dissipadores são elementos que convertem a energia do sistema para um outro tipo de energia, geralmente térmica, que são irrecuperáveis pelo sistema [Silva, 2005]. Ao

contrário das fontes e dos acumuladores, existe apenas existe apenas um tipo desses dispositivos. São elementos cuja formação permitem relacionar a variável de esforço com a variável de fluxo. A equação 4.14 apresenta esse comportamento.

$$e = \phi(f) \tag{4.14}$$

Dissipadores são elementos que não têm capacidade de armazenar energia. Dessa forma, baseado na equação 4.1, a potência instantânea absorvida por eles é definida por:

$$P = ef = \int_0^f e\,df + \int_0^e f\,de \tag{4.15}$$

O primeira integral define a potência absorvida pelo dissipador, enquanto a segunda integral corresponde à potência dissipada no processo. A figura 4.4 apresenta uma representação gráfica dessas componentes.

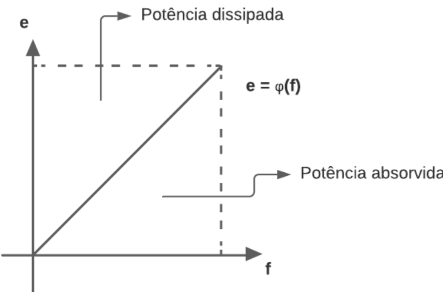


Figura 4.4: Relação entre fluxo acumulado e esforço em dispositivos acumuladores

Tomando como base as definições apresentadas nesta seção, podemos encontrar representações matemáticas para os elementos de todas as outras naturezas [Maitelli and SILVA, 2005] já citadas neste trabalho, relacionando sua variável de esforço com a variável de fluxo. Nas fontes a relação é dada apenas pela quantidade de esforço ou fluxo que ela fornece ao sistema. Os demais dispositivos são representados por sua impedância, ou seja, pelo quociente entre a variável de esforço e a variável de fluxo. A tabela 4.3 mostra as funções matemáticas no domínio de Laplace que representam cada um dos elementos físicos que podem compor um sistema.

Natureza	**F. Esforço**	**F. Fluxo**	**A. Esforço**	**A. Fluxo**	**Dissipador**
Elétrico	V	I	LS	$C^{-1}S^{-1}$	R
Mecânico	v	F	$K^{-1}S$	$M^{-1}S^{-1}$	B^{-1}
Mec. Rot.	w	τ	$K_R^{-1}S$	$J^{-1}S^{-1}$	B_R^{-1}
Fluídico	P	Q	$L_F S$	$C_F^{-1}S^{-1}$	R_F
Térmico	T	q	-	$C_T^{-1}S^{-1}$	R_T

Tabela 4.3: Representações matemáticas dos componentes físicos no domínio de Laplace

4.2.2 Outros componentes físicos

Além dos elementos citados, existem outros tipos de componentes que poderemos considerar. As fontes de energia controladas, de esforço ou de fluxo são dispositivos que atuam da mesma maneira que as fontes ideais apresentadas nas seções anteriores. No entanto, esses dispositivos são dependentes de outras variáveis de modo que a fonte de esforço controlada fornece esforço ao sistema em função valor assumido por uma outra variável de esforço ou de fluxo presente no sistema [Silva, 2005]. O caso é análogo para fontes de correntes controladas. As equações 4.16 e 4.17 mostram o comportamento da fonte de esforço e de fluxo controladas, respectivamente, onde ρ representa uma variável qualquer do sistema.

$$e = \phi(\rho) \tag{4.16}$$

$$f = \phi(\rho) \tag{4.17}$$

Todos os componentes físicos mencionados até agora são os denominados componentes de uma porta, que necessitam apenas de uma conexão de energia para atuar no sistema. Há ainda aqueles que utilizam duas conexões de energia resultando em comportamentos diferentes, mas apresentando características bem interessantes como ferramentas, fazendo assim com que surjam novas possibilidades para soluções de uma maior gama de problemas. Como exemplo desse tipo de dispositivo, podemos citar os transformadores e giradores [Maitelli and SILVA, 2004], por serem os elementos mais básicos que evidenciam tais características.

De modo geral, os elementos de duas portas são capazes de realizar transformação de energia [Maluf, 2020]. Para isso, sua primeira porta deve ser aquela que recebe a energia, para que então a energia transformada seja retransmitida ao sistema pela segunda porta. A equação 4.18 apresenta essa característica.

$$\begin{pmatrix} e_f \\ f_f \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} e_i \\ f_i \end{pmatrix} \tag{4.18}$$

Considerando elementos giradores ideais de ganho G, temos que $a_{11} = a_{22} = 0$, $a_{12} =$ G e $a_{21} = G^{-1}$. Por outro lado, um transformador ideal de ganho N apresenta valores a_{11} = N, $a_{22} = N^{-1}$ e $a_{12} = a_{21} = 0$. [Maitelli and SILVA, 2005].

4.3 Formas de representação de sistemas físicos

Com a evolução da tecnologia, o surgimento de novas formas de representação para sistemas físicos se torna crescente, possibilitando disponibilizar mais ferramentas para que

engenheiros de controle possam elaborar seus projetos com mais precisão. Dentre essas formas de representação, o grafo de ligação e o diagrama de fluxo de sinais são as mais utilizadas por serem representações capazes de representar sistemas físicos de qualquer natureza [El-Hajj and Kabalan, 1995] [Negrão, 2012] permitindo fazer simplificações e aplicar teoremas matemáticos para encontrar equações que os representem.

4.3.1 Grafo de Ligação

O grafo de ligação é uma ferramenta gráfica poderosa capaz de representar sistemas físicos com os mais variados tipos de componentes encontrados nos diversos domínios físicos, principalmente quando em um mesmo sistema é composto por elementos de mais de um domínio diferente. O objetivo principal do grafo de ligação é representar as trocas de energia existentes no sistema. A figura 4.5 mostra alguns exemplos de possíveis grafos de ligação representando sistemas elétricos.

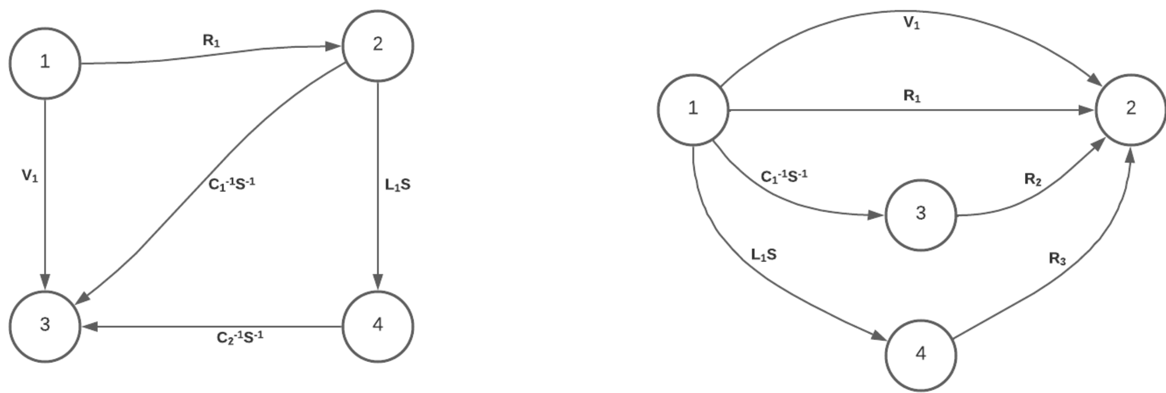


Figura 4.5: Exemplos de grafos de ligação

Esse tipo de representação, por ser um grafo, é composto por vértices e arestas direcionadas, o que o classifica como um dígrafo. Cada vértice indica pontos com mesma quantidade de esforço no sistema [Maitelli and SILVA, 2005]. As arestas representam os componentes, de modo que seu peso é indicado pela expressão matemática que expressa a característica do sistema como visto na seção anterior. Caso um sistema possua elementos de duas portas, esses elementos serão representados por duas arestas diferentes com mesmo peso, sendo uma para cada porta.

Como o diagrama considera a lei de conservação de energia para representar o sistema [Broenink, 1999], a direção das arestas é definido pelo sentido de transferência de potência entre os pontos de esforço. De forma geral, é correto dizer que dentro de um par de vértices, o que possuir menor valor de variável de esforço será a origem, indicando também a direção de fluxo transferido entre os nós.

Dessa forma, é possível reconhecer as propriedades de qualquer sistema físico, obser-

vando o fluxo energético, apenas visualizando um diagrama, implicando em uma interpretação imediata por todos as partes envolvidas no projeto.

4.3.2 Diagrama de Fluxo de Sinais

Enquanto o grafo de ligação trás uma representação gráfica em torno da transferência de energia entre os diversos componentes de um sistema físico, o diagrama de fluxo de sinais nos permite visualizar as relações entre cada uma das variáveis, de esforço e de fluxo do sistema. É um diagrama mais adequado para a obtenção da função de transferência à partir de ambientes computacionais [Nise, 2013].

De modo geral, a ideia do diagrama é estabelecer relações matemáticas entre as variáveis do sistema [Brown, 2001]. Cada vértice representa uma variável e a aresta, ou ramo, é chamada de tramitância e expressa a relação matemática entre a origem e o destino. O valor para cada nó corresponde à soma de todos os ganhos de sinal que chegam até ele.

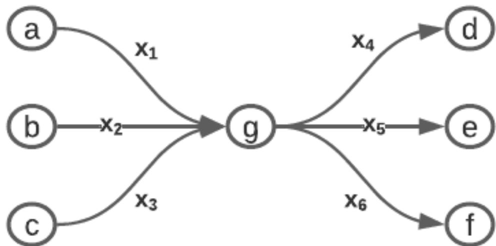


Figura 4.6: Apresentação de componentes e funcionamento de uma DFS

Utilizando a representação gráfica da figura 4.6 é possível extrair as relações matemáticas mostradas na equação 4.19. As variáveis $x_1 ... x_6$ são atribuições de funções de transferência no domínio de Laplace.

$$\begin{aligned} g &= ax_1 + bx_2 + cx_3 \\ d &= gx_4 \\ e &= gx_5 \\ f &= gx_6 \end{aligned} \tag{4.19}$$

À partir desse conceito, diagramas maiores podem ser montados para representar sistemas de qualquer domínio físico, uma vez que as variáveis de fluxo e de esforço são os sinais do sistema. Variáveis de esforço de fontes de esforço e variáveis de fluxo de fontes de fluxo representam vértices denominados variáveis de entrada [Silva, 2005]. Esses vértices não são destinos para nenhuma tramitância, ou seja, é possível apenas a ocorrência

de ligações à partir deles para um outro vértice. Da mesma forma, um nó de saída é caracterizado por possuir apenas arestas que ligue um vértice a ele, nunca sendo ponto de origem para nenhuma outra variável. Na figura 4.6 os vértices "a", "b"e "c"são nós de entrada, enquanto "d", "e"e "f"são os nós de saída. o nó "g"é origem e destino para outros nós, e portanto, chamado de nó misto. A figura 4.7 apresenta um possível modelo de DFS gerado para representar um sistema físico.

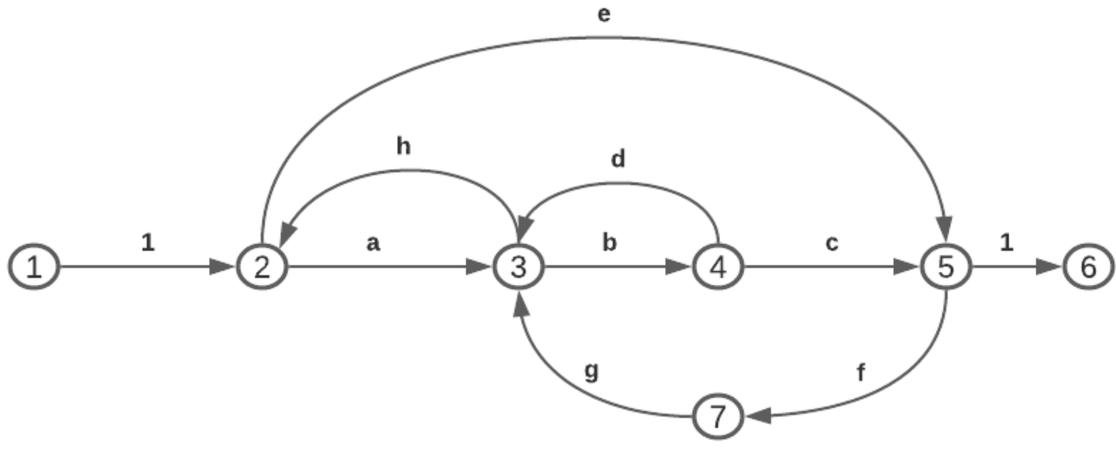


Figura 4.7: Exemplo de DFS

A visualizar a figura, percebemos a formação de vários caminhos, alguns no sentido direto, isto é, partindo da variável de entrada com destino à de saída, e outros no sentido contrário. Caminhos que permitem alcançar a variável de saída à partir da variável de entrada são chamados caminhos diretos. Quando um nó é visitado mais de uma vez, então encontramos um ciclo, ou laço, no diagrama. Para cada um desses elementos, é possível obter seu ganho multiplicando os pesos de cada ramo existente no caminho ou laço. Os vértices 3,4,3 na figura 4.7 representam um ciclo com ganho bd, enquanto 1,2,3,4,5,6 é um caminho direto com ganho abc.

A ideia geral desta ferramenta é possibilitar a realização de simplificações matemáticas até obter o modelo que relaciona diretamente a variável de entrada com a variável de saída. É o mesmo processo quando resolvemos equações lineares e vamos substituindo as variáveis de uma equação por uma expressão que a represente. Os ramos em série ou em paralelo vão sendo substituídos por ramos equivalentes.

4.4 Regra de Mason

Apesar de bastante útil, o processo de simplificação nos diagramas de fluxo de sinais pode ser bastante custoso se executado à partir de um sistema computacional [Nise, 2013], fazendo com que a sua aplicação não seja viável no desenvolvimento de *softwares*. Assim,

um procedimento matemático se fez necessário para que, em conjunto com a DFS, o processo de modelagem de sistemas pudesse ser mais efetivo, onde a aplicação de conceitos da teoria dos grafos para efetuar o processo de leitura em um diagrama de sinais, tornasse possível chegar a resultados precisos, exigindo um menor poder de processamento. Isso possibilitaria processar modelos maiores e mais complexos.

A regra de Mason surge como uma solução ao problema apresentado. É um algoritmo matemático utilizado para obter a função de transferência, dado uma variável de entrada e uma variável de saída, à partir do DFS de um sistema. A equação 4.20 apresenta sua definição.

$$G(s) = \frac{C(s)}{R(s)} = \frac{\sum_k T_k \Delta_k}{\Delta} \tag{4.20}$$

Onde:

- **G(s):** Função de transferência que representa o comportamento do sistema à partir de uma variável de entrada e uma variável de saída
- **k:** Cada um dos caminhos diretos entre o nó de entrada e o nó de saída
- T_k**:** Ganho do k-ésimo caminho direto
- Δ**:** 1 - Somatório dos ganho dos laços de primeira ordem + Somatório dos ganhos dos laços de segunda ordem - Somatório dos ganhos dos laços de terceira ordem + Somatório dos ganhos dos laços de quarta ordem - $\cdots$
- Δ_k**:** Diferença entre o valor de Δ e somatório dos valores de ganho dos laços que tocam o k-ésimo caminho direto entre os nós de entrada e saída, ou seja, possuem vértices em comum.

A ordem dos laços é definida pelo agrupamento de ciclos que não se encontram, isto é, não possuem nenhum vértice em comum. Dessa forma, se os laços A, B e C apresentam essa característica, AB, AC e BC seriam laços de segunda ordem, e ABC seria um laço de terceira ordem.

Capítulo 5

Metodologia

Neste capítulo serão apresentados os procedimentos, dificuldades e algoritmos utilizados para a implementação deste trabalho. Foi desenvolvida uma aplicação de construção e simulação de sistemas lineares a fim de obter a função de transferência que define o comportamento do modelo, dado um ponto de referência para a entrada e outro para saída. Com isso é possível simular e analisar um projeto inicial de controle de sistemas antes de iniciar sua fase de construção. A aplicação busca uma abordagem detalhada dos procedimentos executados, proporcionando a visão do grafo de ligação, das matrizes de relacionamento dos componentes, e do diagrama de fluxo de sinais resultantes, que são fundamentais para a obtenção da equação final. Dessa forma, o usuário poderá entender como se dá o comportamento de cada elemento, seja ele de natureza elétrica, física ou hidráulica. Além disso, houve uma grande preocupação em relação à disponibilidade da aplicação quanto ao seu grau de poder computacional, desde que para cumprir os objetivos propostos, o resultado deste trabalho deveria estar disponível para usuários que não possuam um computador com sistema operacional compatível como normalmente exigem as aplicações de simulação e processamento de cálculos numéricos. Por isso, optou-se pela alternativa do desenvolvimento Web, de modo que os recursos pudessem ser acessados através de um navegador.

O objetivo do AutoTF é seguir um fluxo que se inicia com a construção do desenho do sistema a ser analisado. Caso não sejam encontradas pendências ou falhas no projeto, serão mostrados o grafos e matrizes resultantes, para que, por fim, possa ser apresentado o resultado final através da função de transferência. Dessa forma, o processo de implementação do projeto foi dividido em três etapas: Tela Inicial, Painel de Resultados Parciais e Apresentação da Função de Transferência.

A tela inicial é onde o usuário poderá realizar o desenho do sistema físico, tendo à sua disposição uma paleta de componentes e ferramentas para manuseá-los como desejar dentro da área de desenho. É o ambiente que terá mais possibilidades de interação com o operador, sendo necessário pensar com cautela a respeito de quaisquer eventos que possam ser disparados.

O painel de resultados parciais, apesar de ter muitas visões disponíveis, todas seguem o mesmo padrão. De modo geral, a interface gráfica desta etapa estará dividida entre apresentação de grafos e apresentação de matrizes, fazendo com que as telas com o mesmo tipo de visão apresentem características semelhantes.

Por outro lado, a tela de apresentação da função de transferência é a que apresenta menor complexidade em sua interface gráfica. O objetivo dela é apenas obter do usuário a informação de variáveis de entrada e saída nas quais ele deseja ver o resultado, e exibir as equações polinomiais resultantes.

A construção das duas últimas etapas foram realizadas em uma janela interna do sistema web denominada *modal*, fazendo com que o painel de resultados parciais fique sobreposto à tela inicial, e a tela da função de transferência sobreposta ao painel. Dessa forma é possível obter uma maior velocidade de transição entre as etapas do AutoTF, evitando que o usuário aguarde carregamentos desnecessários.

Como objetivo é apresentar uma solução que seja disponibilizada em ambiente Web, o código foi escrito na linguagem Javascript, utilizando a fundo a API do Canvas para inserir desenhos dinâmicos dentro de um contexto HTML, sem nenhum tipo de conexão à serviços externos para armazenar ou consultar dados. Todos os recursos que são utilizados, são reservados pelo próprio navegador do usuário e uma vez que a janela seja fechada, todo o trabalho será perdido caso não tenha sido salvo. A seguir, entraremos em detalhes à respeito da implementação de cada uma das etapas do projeto, relatando todas as estratégias adotadas, desafios e resoluções até chegar ao estado de apresentação desta versão do AutoTF.

5.1 Tela Inicial

O primeiro passo para a construção da tela de apresentação do AutoTF foi definir como seriam dispostos os elementos da modelagem na tela, de forma que o usuário pudesse interagir, movendo, selecionando, rotacionando ou apagando o objeto da maneira que achasse necessário. O Canvas foi escolhido para executar esse papel por proporcionar essa dinâmica. Basta apenas que um algoritmo seja construído para tratar de todos esses cenários. O seu uso permite que formas geométricas, texto e imagens sejam inseridas dentro de um espaço livre de maneira estática, ou seja, uma vez que o desenho foi colocado em tela, não é possível fazer com que ele se mova. No entanto, aliado ao poder computacional do Javascript é possível implementar uma atualização de *frames*, de modo que todo o conteúdo seja apagado e redesenhado logo em seguida, em frações de segundo, passando ao usuário a impressão de que o objeto está se movendo.

5.1.1 Estrutura dos componentes

Uma vez definida a base inicial para desenho do projeto, foi preciso saber qual seria a melhor alternativa para representar os elementos. Desenhar a forma vetorizada do elemento no canvas representaria um trabalho desnecessário considerando que cada elemento deve estar representado a partir de quatro diferentes níveis de rotação, sendo necessário um desenho diferente para cada uma delas. Dessa forma optou-se por utilizar imagens em boa resolução de cada um dos componentes envolvidos. Essas imagens são cadastradas no arquivo HTML com exibição oculta, para que o canvas utilize sempre que for indicado.

Tabela 5.1: Representações gráficas dos componentes elétricos

Componente	**Representação Gráfica**
Resistor	
Capacitor	
Indutor	
Fonte de Tensão	
Fonte de Corrente	

As imagens apresentadas na tabela 5.1 mostram a representação gráfica inicial para cada um dos componentes existentes no AutoTF. Para cada elemento deve ser associado a imagem correta no momento de construir sua representação gráfica no canvas.

Vale destacar que são necessárias quatro imagens do mesmo componente, uma para cada configuração de rotação. O projetista poderá escolher girar um objeto específico em 0, 90, 180 e 270 graus. Além disso, ao adicionar uma imagem ao canvas, atributos de posicionamento e dimensionamento são essenciais para definir de que forma o elemento será mostrado ao usuário, servindo como parâmetros para a implementação da função de desenho. Alguns outros parâmetros que controlam o comportamento de cada objeto dentro do canvas também são necessários para que ele possa ser manipulável, sem falar da necessidade da criação de conectores, que precisam ser associados aos componentes para que possam ser feitas conexões entre eles. e do armazenamento de atributos matemáticos para que a solução possa considerar os objetos adicionados em cálculos posteriores. Visando essas questões pensou-se então em criar uma estrutura de dados para armazenar todas esses atributos, de forma a prezar pela boa organização no código fonte do AutoTF ao encarar essa gama de atributos propostos para realização das funções de desenho do sistema.

É notavel que todos os elementos de uma mesma natureza possui atributos semelhantes no contexto deste trabalho. Entretanto, como vimos na tabela 5.1, mesmo que as imagens

possuam formatos parecidos, elas se diferenciam quanto à suas proporções dimensionais. Além do mais, cada um dos diferentes tipos de componentes possui imagem, unidades e funções de transferência distintas, ou até mesmo valores iniciais diferentes à depender de sua característica física. Enquanto 20V é um valor comum para fontes de tensão, uma fonte de corrente com 20A se torna um cenário bem mais improvável.

Surgiu então a necessidade de organizar as diferentes estruturas propostas para os componentes em um modelo de herança, onde todas as classes referentes à componentes físicos herdassem de uma classe *Component* que implementaria todos esses atributos mencionados no parágrafo anterior e funções de comum execução à todas os elementos. Na figura 5.1 é possível visualizar como a estrutura ficou definida.

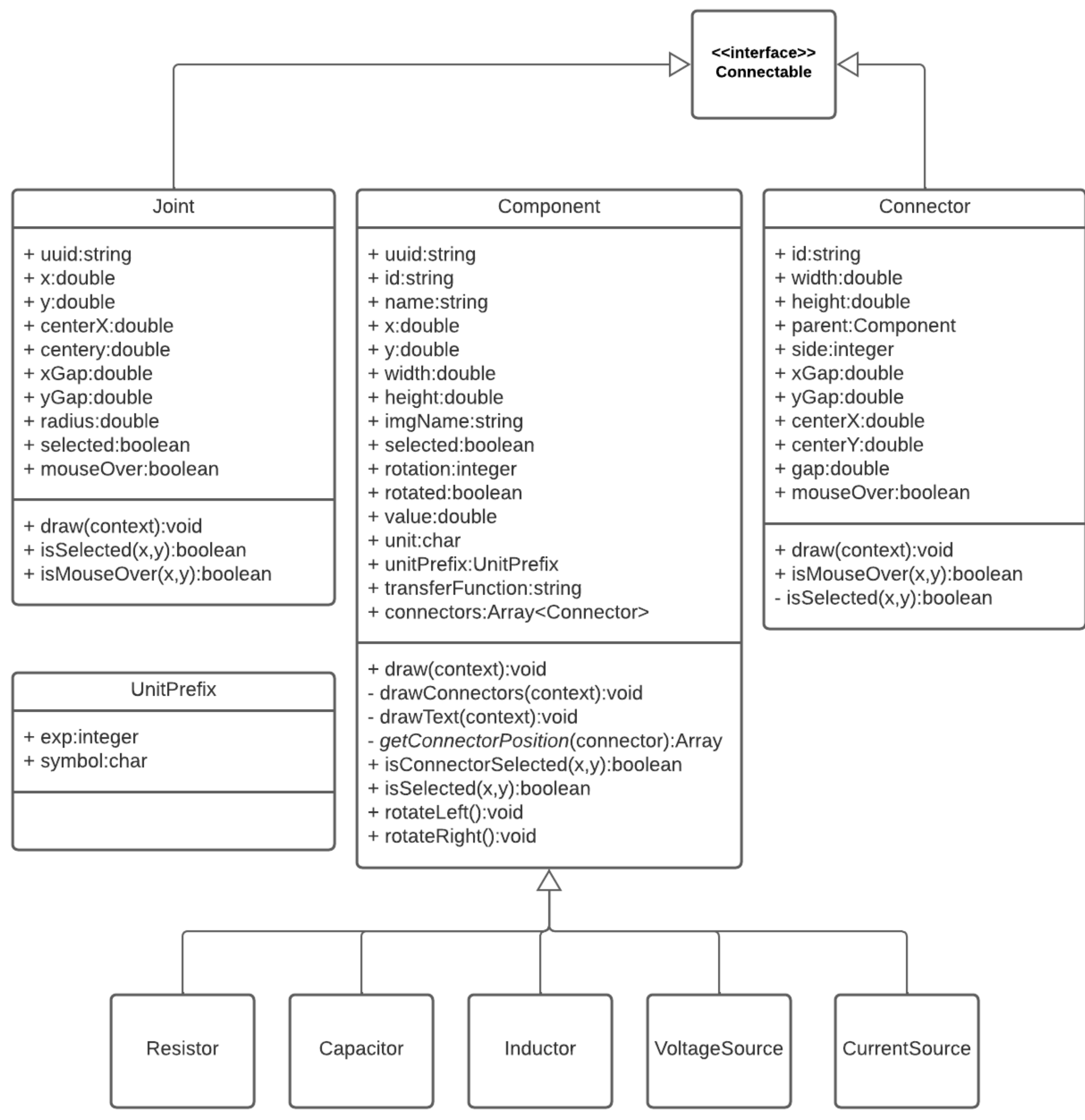


Figura 5.1: Diagrama de Classes dos componentes disponíveis para desenho.

As classes *Component*, *Joint* e *Connector* são as principais na etapa de modelagem do desenho gráfico do sistema. Uma outra classe *Connection* também é de fundamental importância para a construção do protótipo do sistema físico, mas deixaremos para falar sobre ela na próxima seção. No diagrama foram representadas apenas as principais funções, que são chamadas para execução de ações diretas nos objetos desenhados. Vale lembrar que o conceito de orientação à objetos está sendo amplamente utilizado neste trabalho, então cada objeto deve guardar atributos e desempenhar uma ação que faz referência à si mesmo. Podemos citar como exemplo o método *draw*, responsável por desenhar a representação gráfica do elemento instanciado no canvas. Cada elemento pode possuir uma sequência diferente de execução nesta função. Por isso ela deve existir em todas as principais classes. Ao chamar uma função *draw* de um determinado componente herdeiro da classe *Component*, é identificado à partir do conceito de polimorfismo, que existe uma implementação específica para ele, que será seguida de acordo com a classe filha correspondente.

As tabelas 5.2 e 5.3 descrevem os atributos e funções mostrados no diagrama de classe.

Atributo	**Descrição**
uuid	Token único de identificação do componente
id	Identificador do componente que será utilizado no grafo de ligação
name	Identificador do tipo de componente. Ex: Resistor
x	Coordenada x de posicionamento do componente no canvas
y	Coordenada y de posicionamento do componente no canvas
width	Largura da imagem do componente desenhada no canvas
height	Altura da imagem do componente desenhada no canvas
imgName	Nome do elemento HTML que guarda a imagem do componente
selected	*Flag* indicadora de componente selecionado
rotation	Indicador de rotação do componente
rotated	Flag indicadora de rotação do componente
value	Valor numérico do componente
unit	Unidade de medida
unitPrefix	Múltiplo da unidade de medida
transferFunction	Valor da impedância no domínio da frequência
connectors	Lista de conectores do componente

Tabela 5.2: Atributos da classe *Component*

A partir desse modelo, foi possível estabelecer um padrão de comportamento para todos os componentes. Quando sua função *draw* é chamada, o primeiro passo será identificar a rotação atual do componente a partir da variável *rotation*. O seu valor definirá para qual lado ficará o conector positivo de cada componente. A figura 5.2 exemplifica como são formados os valores das variáveis *rotation* e *rotated*.

Após definir qual será o grau de rotação aplicado no componente, são estabelecidas variáveis internas para armazenar as larguras reais do componente. Caso o valor do

Função	Descrição
draw	Desenha o componente no canvas
drawConnectors	Chama a função draw de cada um de seus conectores
drawText	Escreve o conteúdo textual (*id*, *value*, *unit*) no canvas
getConnectorPosition	Calcula a posição de um conector específico
isConnectorSelected	Verifica se o mouse está acima de algum dos conectores do componente, marcando-o como selecionado
isSelected	Verifica se o mouse está acima do componente, marcando como selecionado em caso positivo
rotateLeft	Incrementa o indicador de rotação ou define como 1 se o valor anterior for 4
rotateRight	Decrementa o indicador de rotação ou define como 4 se o valor anterior for 1

Tabela 5.3: Funções da classe *Component*

atributo *rotated* seja *true*, a imagem é desenhada com os valores para largura e altura trocados. Além disso, o texto será posicionado, pela função *drawText* à esquerda do componente ao invés do topo caso o componente esteja rotacionado. É verificado se a *flag selected* está marcada para que seja desenhado um retângulo azul ao redor do componente indicando a seleção do mesmo, e por fim, é chamada a função *draw* para cada conector do componente através do método *drawConnectors*.

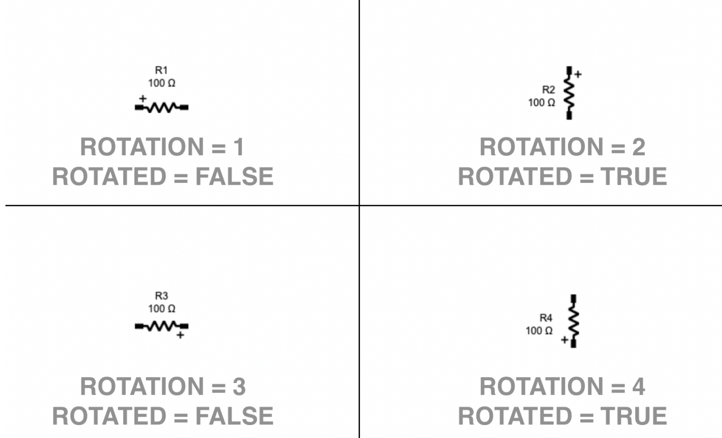


Figura 5.2: Configurações de rotação disponíveis para os componentes

Todo esse procedimento acontece no momento em que são gerados os componentes e conexões do sistema, em um processo de atualização de *frame* disparado pelo contexto principal do AutoTF.

5.1.2 Conectores e juntas

Uma vez definida as regras de posicionamento dos componentes, foi necessário saber como fazer para interligá-los uns com os outros. Para que pudesse ser estabelecido pontos de

origem e destino para as conexões, deveriam ser definidas novas estruturas para controlar esse procedimento. Sendo assim, saber quais estruturas seriam necessárias, dependeria da criação de padrões para conexão entre componentes, já implementando rotinas de execução para evitar que o usuário realizasse procedimentos incorretos. Partimos então de dois princípios para construção das novas estruturas de dados:

1. Cada conector pode ter apenas uma única conexão associada a ele
2. Um conector não pode ser ligado à si mesmo

Sendo assim, foi necessária a criação de mais uma estrutura de dados que possibilitaria múltiplas conexões permitindo criar o efeito de paralelismo nos circuitos. A junta é um conector que possui algumas características de componente, já que pode estar de forma independente inserida na área de desenho. Sua particularidade é ser o único tipo de elemento que pode e deve ser associado a mais de uma conexão.

Na figura 5.1, podemos perceber que as classes *Connector* e *Joint* são baseadas em uma interface chamada *Connectable*, indicando que são elementos passíveis para associar a uma conexão. Se repararmos em seus atributos e métodos notaremos também que apresentam semelhanças em suas estruturas de classe. A diferença entre um conector e uma junta é a dependência que um conector tem de seu componente "pai". Isso influencia diretamente no posicionamento desses elementos, na estratégia adotada pra criar linhas de conexão, e posteriormente, no algoritmo para geração do grafo de ligação.

Ambas as classes possuem as mesmas funções, tendo como única novidade a função *isMouseOver* que atuará da mesma forma que o método *isSelected*, mudando apenas o momento de sua execução. Ela é necessária para indicar, na interface gráfica, no modo de criação de conexões, que o elemento é selecionável e pode ser conectado, comportamento que contribui para a questão da usabilidade citado anteriormente. As tabelas 5.4 e 5.5 apresentam as variáveis existentes nas classes *Connector* e *Joint*.

Uma junta é graficamente representada no Canvas como uma circunferência preenchida. Está disponível na paleta de componentes podendo ser selecionada, movida e excluída. Como não se trata de uma figura retangular, não é possível aplicar qualquer tipo de rotação à ela.

Por outro lado, o conector é um elemento criado em conjunto com o seu componente. é representado por um retângulo, e suas coordenadas e dimensões são dependentes do atributo *rotated* de seu componente "pai". Para cada tipo de componente existe uma combinação diferente para obter as coordenadas dos conectores através da função *getConnectorPosition* da classe *Component*. Essa é uma função abstrata que precisa ser reimplementada em cada uma de suas classes filhas.

No momento da geração desses elementos na área de trabalho do sistema, suas variáveis são inicializadas e o cálculo de suas posições são realizados logo em seguida. O caso da

Atributo	**Descrição**
id	Identificador do conector. Indica também sua polaridade (A - negativo, B - positivo)
width	Largura do objeto na representação gráfica
height	Altura do objeto da representação gráfica
parent	Componente gerador
side	Indicativo de posição indicando em qual borda do componente o conector está localizado. 1- Direita, 2- Base, 3- Esquerda, 4- Topo
xGap	Espaço adicional no eixo X, utilizado como referência pelas conexões
yGap	Espaço adicional no eixo Y, utilizado como referência pelas conexões
centerX	Cordenada X do ponto de conexão
centerY	Cordenada Y do ponto de conexão
gap	Valor adicionado às coordenadas x e y, para obter xGap e yGap
mouseOver	*Flag* para indicar quando o mouse sobrepõe o conector

Tabela 5.4: Atributos da classe *Connector*

Atributo	**Descrição**
uuid	Token único de identificação do componente
x	Coordenada x de posicionamento da junta no canvas
y	Coordenada y de posicionamento da junta no canvas
centerX	Coordenada X do ponto de conexão
centerY	Coordenada Y do ponto de conexão
xGap	Espaço adicional no eixo X, utilizado como referência pelas conexões
yGap	Espaço adicional no eixo Y, utilizado como referência pelas conexões
radius	Raio da junta utilizado na representação gráfica
selected	*Flag* indicadora de junta selecionada
mouseOver	*Flag* para indicar quando o mouse sobrepõe o conector

Tabela 5.5: Atributos da classe *Joint*

junta é o cenário mais simples, onde os valores para x e y representam o centro da circunferência que a representa, portanto, os valores de *centerX*, *centerY*, *xGap* e *yGap* já estão definidos. Já para o conector, após receber de seu componente as variáveis x e y onde devem estar localizados, os atributos de *center* e *gap* precisam ser calculados para consulta posterior do processo de geração de uma conexão.

As funções *draw* destas classes são de implementação trivial, que se resumem em verificar a rotação do componente, no caso do conector, calcular as coordenadas e desenhar a forma geométrica de acordo com os tamanhos definidos para altura e largura ou raio. O processo de rotação dos componentes influenciam diretamente nos conectores, que também devem estar rotacionados, e portanto, com valores para altura e largura invertidos. Caso o elemento esteja selecionado, a cor da linha e do preenchimento é alterada de preto para azul. Uma particularidade na função de desenho destas classes, é que além de ter que verificar se o componente está selecionado, deverá ser checado também se a *flag*

mouseOver está marcada. Em caso positivo, um sombreamento circular na cor azul claro, com área pouco maior do que a ocupada pelo componente, deve ser inserido para mostrar ao usuário que o elemento pode ser clicado no modo de criação de conexões. Para que o sombreamento não sobreponha o componente já desenhado, deve ser alterado o atributo de composição do canvas para *destination-over*, para que a nova forma fique abaixo de todas as outras já desenhadas, e em seguida o valor do atributo é retornado para *source-over*.

5.1.3 Conexões

Conexões são estruturas que conectam componentes, através de seus conectores, e juntas. É uma das entidades mais importantes nesta primeira fase do AutoTF, com a implementação mais complexa dentre todas as outras já citadas nesse capítulo. O conceito básico utilizado para criação de conexões pode ser resumido em uma única tarefa: a geração de pontos intermediários que ao serem ligados uns aos outros, formam retas que representarão uma conexão. Dessa forma, podemos dizer que um objeto Conexão é formado por um conjunto de entidades denominados "Linhas de Conexão".

Para que uma nova conexão seja estabelecida, é necessário que sejam informados os pontos de origem e destino pertencentes a ela. A partir disso, uma série de decisões devem ser tomadas para adicionar pontos intermediários a fim de que em termos de interface gráfica sejam obtidas boas visualizações de conexão entre os dois elementos, prezando pela não colisão entre conexões e componentes, a inexistência de linhas diagonais e a formação do menor caminho possível para evitar que algum espaço seja consumido na área de desenho de forma desnecessária. A nível de cálculo matemático, todo esse tratamento se torna irrelevante, visto que seria preciso apenas saber quais elementos estão conectados entre si, no entanto, levando em conta a preocupação com a aparência do AutoTF, tornou-se um procedimento necessário.

Para falarmos a respeito do algoritmo de definição de pontos, precisamos primeiramente conhecer a formação das classes *Connection* e *ConnectionLine*, sendo essa última mais importante na implementação do controle de colisão, abordado na próxima seção. A figura 5.3 apresenta o diagrama de classes das entidades envolvidas no processo de criação das conexões e a tabela 5.6 descreve cada um dos atributos da classe *Connection*.

Atributo	Descrição
uuid	Token único de identificação do componente
connectorA	Elemento (Conector ou Junta) que representa o ponto de origem
connectorB	Elemento (Conector ou Junta) que representa o ponto de destino
connectionPoints	Lista de pontos intermediários que formarão a conexão
connectionLines	Conjunto de linhas formadas pela ligação entre os pontos intermediários

Tabela 5.6: Atributos da classe *Connection*

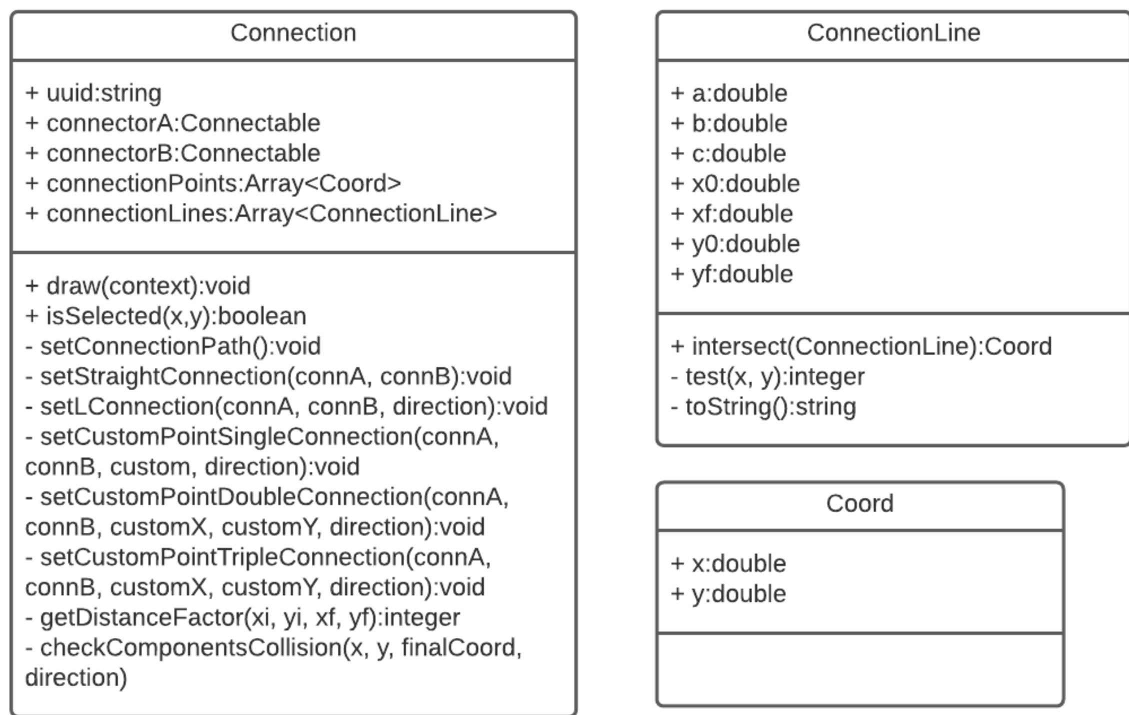


Figura 5.3: Diagrama de Classes das conexões

A implementação de uma conexão se inicia com a inicialização de suas variáveis, marcando o reconhecimento dos pontos de origem e destino do novo objeto. A partir disso, toda vez que o contexto da aplicação precisar atualizar a área de desenho, todos os pontos intermediários devem ser gerados ou recriados para que as novas condições de posicionamento entre os objetos sejam consideradas. Todo esse processo se inicia na função *setConnectionPath*. O primeiro passo é identificar se dentre os elementos que estão sendo conectados existe alguma junta. Em caso positivo, deve ser garantido que o elemento destino da conexão será a junta, por questões de simplificação lógica. Caso as variáveis de origem e destino sejam formadas por conectores, a prioridade para o ponto de origem da conexão será para o conector com id "B"(terminal positivo). Vale lembrar que todas classes que implementam a interface *Connectable* possuem as variáveis *centerX* e *centerY* que serão utilizadas como coordenadas de referência pelas conexões. Dessa forma, após definir qual será o elemento de origem, os valores do primeiro item do array *connectionPoints* serão definidos por essas variáveis.

Em seguida, é necessário verificar a configuração de posicionamento dos dois elementos a serem conectados. Para isso, também são utilizadas as coordenadas de ponto central das juntas ou conectores. Utilizaremos as nomenclaturas dos pontos cardeais para denominar os possíveis cenários, tomando como ponto central o elemento de origem da conexão. Considerando "A"como conector de origem e B como conector destino, e o sistema de coordenadas do Canvas com variável ordenada (y) invertida, temos os seguintes possíveis

cenários:

- A[x] = B[x] e A[y] > B[y], conexão à norte
- A[x] = B[x] e A[y] < B[y], conexão à sul
- A[x] < B[x] e A[y] = B[y], conexão à leste
- A[x] > B[x] e A[y] = B[y], conexão à oeste
- A[x] > B[x] e A[y] > B[y], conexão à noroeste
- A[x] > B[x] e A[y] < B[y], conexão à sudoeste
- A[x] < B[x] e A[y] > B[y], conexão à nordeste
- A[x] < B[x] e A[y] < B[y], conexão à sudeste

Para cada um desses cenários, devem ser consideradas a existências de elementos do tipo *Joint* e as possíveis atribuições da variável *side* descritas na tabela 5.4 para todos os objetos da classe *Conector* envolvidos. Levando em conta que a variável *side* pode assumir quatro valores diferentes, e também o pior cenário, em que os dois elementos envolvidos são da classe *Connector*, teremos 8 possibilidades para cada um dos pontos cardeais previamente definidos. Cada um desses casos foram analisados separadamente, aplicando um padrão de conexões previamente estabelecido.

Os tipos de conexões foram classificados quanto à sua quantidade de linhas, de forma que foi preciso estabelecer padrões de conexões com até 5 linhas para atender todas as possibilidades mostradas anteriormente. Além disso, foi importante observar a orientação da primeira e última reta da conexão para elaborar a estratégia para cada cenário. Conexões com número par de linhas conseguem interligar apenas conectores que estejam em orientação oposta, ou seja, um deles com saída à partir da vertical e o outro com entrada pela horizontal, ou vice versa. Essa denominação deve existir para que a conexão não entre em rota de colisão com o componente que a originou. Nesse sentido, conexões com número ímpar de linhas podem ligar apenas conectores com mesma orientação entre si. Um outro ponto importante é a inclusão do elemento do tipo *Joint* nesse cenário, que pode assumir entradas e saídas tanto pela horizontal como pela vertical, visto que o elemento não possui área de colisão com as conexões. A tabela 5.7 mostra as possíveis formas para cada uma das categorias de conexão estabelecidas.

O nível de prioridade para a escolha de um dos tipos de conexão para definição dos pontos intermediários foi definido para que existam um menor número possível de linhas ligando os dois elementos. Caso um determinado tipo não atendesse a necessidade exibindo um bom resultado gráfico ao final do processo, eram adicionadas duas linhas para

Tabela 5.7: Possíveis tipos de conexão entre componentes

Tipo Conexão	**Ligação**	**Principais Formatos**
Conexões retas	Mesma Orientação	
Conexões "L"	Orientação Oposta	
Ligações 3 linhas	Mesma Orientação	
Ligações 4 linhas	Orientação Oposta	
Ligações 5 linhas	Mesma Orientação	

então realizar uma nova análise. Esse foi o processo mais custoso na construção do AutoTF, onde técnicas mais avançadas para tomada da decisão de escolha do tipo de conexão empregado não foram implementadas à fim de que pudessem ser obtidos resultados mais imediatos em relação ao propósito final desta solução. Algoritmos de busca poderiam ser implementados em versões posteriores para facilitar esta etapa, corrigindo talvez até possíveis cenários não mapeados no desenvolvimento deste trabalho.

A formação dos tipos de conexão são realizadas pelas funções *setStraightConnection* para conexões retas, *setLConnection* para conexões "L", *setCustomPointSingleConnection* para ligações com 3 linhas, *setCustomPointDoubleConnection* para ligações com 4 linhas e *setCustomPointTripleConnection* para ligações com 5 linhas. Todas elas se resumem na adição de pontos intermediários à lista *connectionPoints*, inclusive da última coordenada que marca o destino da conexão. Nelas são implementadas também o controle de colisão com componentes, que falaremos mais adiante. Para todas as funções de tipo de conexão são recolhidas as variáveis contendo elemento de origem e destino, além de uma *flag* chamada *direction*. É nela que será indicado se o primeiro ponto intermediário vai ter uma variação de posição no eixo x (*position = 0*) ou y (*position = 1*), a depender da orientação do componente de saída. A única exceção para essa regra é a *conexão de uma linha (reta)* que possui apenas as coordenadas de origem e destino em sua lista de pontos.

Para tipos de conexões com mais de duas linhas, atributos de posicionamento personalizados são obrigatórios para que seja definido os tamanhos das linhas intermediárias do caminho até chegar ao destino. A figura 5.4 mostra como cada variável atua na composição do caminho da conexão.

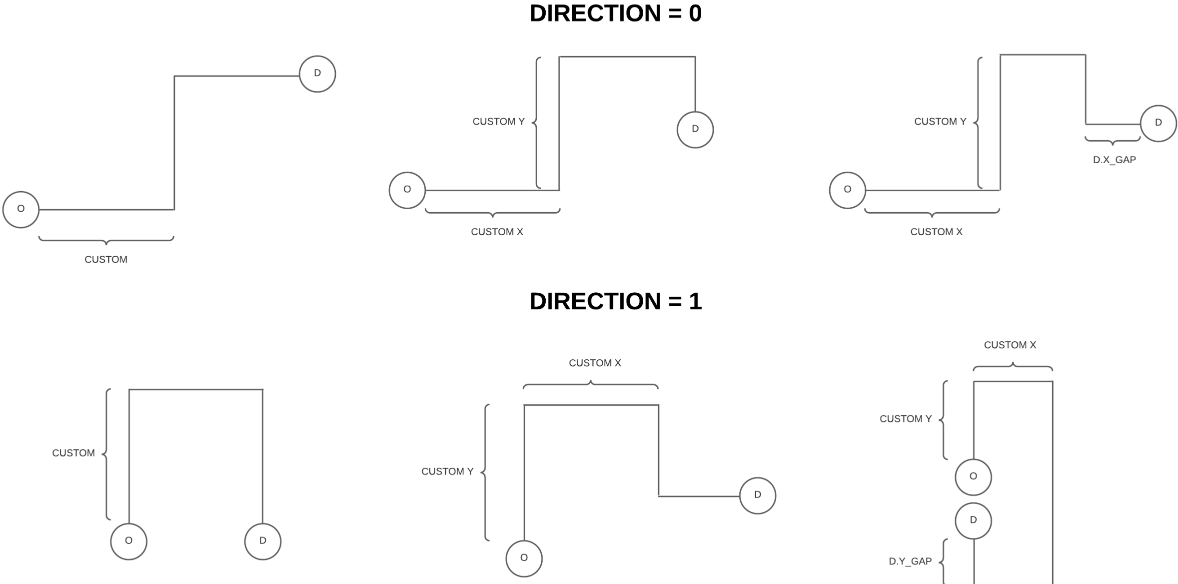


Figura 5.4: Variáveis personalizadas para definição dos pontos intermediários das conexões

De forma geral, podemos ver que as linhas manipuláveis são sempre as primeiras à

partir da origem. Nas conexões com 3 linhas é preciso apenas de um parâmetro adicional que vai definir onde o segundo ponto vai estar localizado, seja variável na horizontal ou na vertical. A partir de 4 linhas, já são necessários dois parâmetros, um para o segundo ponto e outro para o terceiro. A ordem em que eles serão utilizados depende diretamente do atributo *direction*, como mostrado na figura 5.4. Nas conexões com 5 linhas, é necessário utilizar o atributo *xGap* (ou *yGap*) definido nos objetos do tipo *Connectable*. Utilizando essas configurações foi possível criar todas as formas de representação gráfica existentes no AutoTF.

Com todas as estratégias apresentadas, foi possível então definir como seria montada a função *draw* da classe *Connection*. O procedimento se torna bem simples quando se já tem todos os pontos necessários da conexão. Basta percorrer a lista de pontos, chamando a função de desenhar linhas até cada uma das coordenadas. A verificação de seleção também ocorre para as conexões, fazendo com que a cor da linha mude de preto para azul caso o atributo *selected* esteja marcado como *true*. O próximo passo é percorrer todas as equações de retas que formam todo o percurso para identificar colisões com outras conexões. Dessa forma, a função *draw* adiciona imagens de *bypasses* para indicar que aquela intersecção não se trata de uma junta, e as linhas não se tocam.

5.1.4 Controle de Colisão

Com o desenho dos componentes e suas conexões já desenvolvidos, o objetivo passou a ser organizar os elementos na tela para que mesmo com interações incorretas realizadas pelo usuário, a legibilidade do circuito seja preservada ao máximo. Para esse fim, foram identificadas dois recursos que poderiam ser implementados para contribuir com esse processo. O controle de colisão componente-conexão e o controle de colisão entre conexões. Esses dois recursos devem ser executados durante a geração dos pontos e montagem do desenho das conexões existentes no projeto. Dessa forma, verificações devem ser realizadas identificando todos os pontos de intersecção entre esses elementos.

Colisão componente-conexão

A colisão entre uma conexão e componentes presentes na área de desenho acontece ao desenhar retas intermediárias que entram em contato com a área retangular ocupada por qualquer elemento.

A detecção da colisão acontece por meio da função *checkComponentsCollision* da classe *Connection*. Ela é chamada antes de estabelecer alguns pontos intermediários que são propícios a gerar colisões. Sua variável *direction* possui a mesma função apresentada nas funções de construção de tipos de conexão. Seu propósito atual é indicar a orientação da reta a ser formada para que o algoritmo saiba qual é a coordenada variante. Sendo assim, para cada componente verifica-se primeiramente se sua região cobre algum ponto

da coordenada não variante da reta. Em caso negativo, é afirmado que não existe colisão com aquele componente, pois mesmo que a coordenada variante assumisse valores iguais aos preenchidos pelo componente, eles ainda não se tocariam. Em caso positivo, deve ser realizada uma nova verificação em torno da coordenada variante e o valor de *finalCoord*. Para isso, verifica se cada uma das duas extremidades do componente está entre o valor da coordenada variante e de *finalCoord* (ou o contrário, caso *finalCoord* < coordenada variante). As extremidades do componente são definidas por x e x + largura caso *position* = 0 e y e y + altura caso *position* = 1. Essa estratégia foi adotada para que não fosse preciso comparar ponto a ponto da nova reta, procurando saber se ela estaria contida ou não no componente. A imagem 5.5 exemplifica esse cenário.

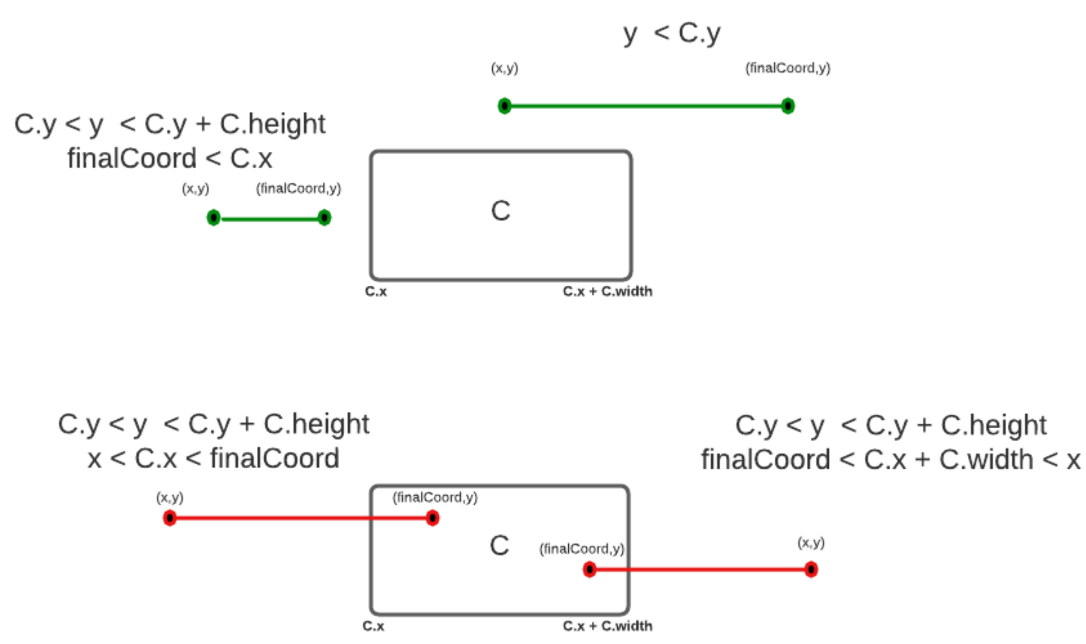


Figura 5.5: Cenários de colisão conexão-componente.

Uma vez identificado o estado de colisão, o comportamento realizado para contornar a situação depende do tipo de conexão que está tentando se construir. Em alguns casos, é necessário alternar o tipo para algum que possua maior número de retas, ao se deparar com casos em que não restam possibilidades de completar a ligação entre os elementos com o caminho original. Para conexões de uma linha, caso exista um cenário de colisão com um terceiro elemento, a representação gráfica passará a assumir o formato de uma conexão de 5 linhas, a fim de desviar o percurso do componente que está no meio do caminho. O mesmo acontece com as conexões "L", que ao serem impedidas de chegar ao destino por um outro componente, se transforma em uma conexão de 4 linhas.

As conexões de 3 linhas em diante são tratadas de maneira diferente. A função *get-*

DistanceFactor é chamada para calcular um fator de variância para ser adicionado às variáveis *custom*. Dentre as suas variáveis, *xi* e *yi* representam o ponto de origem da próxima reta formada em provável colisão, e *xf* e *yf* o ponto destino. Dessa forma, é implementado um algoritmo de busca, que procura o fator com menor valor absoluto até que próxima reta gerada à partir do valor da coordenada *custom* não esteja em rota de colisão. Na conexão com 3 linhas por exemplo, a segunda reta, partindo da origem é aquela que será verificada o estado de colisão. A busca do fator é efetuada de modo que sejam alternados os valores entre positivo e negativo e incrementando a 1 após testar as duas possibilidades de sinal para o mesmo valor. Logo, os valores iniciais testados são: 0, -1, 1, -2, 2, -3, 3, e assim por diante. A função *checkComponentsCollision* é chamada a cada iteração para verificar se a reta verificada colide com algum componente.

Colisão entre conexões distintas

A verificação de colisão entre conexões distintas acontece durante a execução da função *draw* da classe *Connection*. Em sua primeira implementação, pensou-se em um modelo de análise por força bruta, em que cada ponto das retas de um componente dentro de um intervalo de 1 unidade entre eles seria verificado com todos os pontos de todas as retas dos outros objetos *Connection* dentro da lista do contexto geral do AutoTF. Mas logo a ideia foi desmanchada ao perceber que as interações com o sistema começavam a ficar mais lentas. Apesar de o Javascript ser uma linguagem poderosa, que consegue trabalhar com um grande número de informações, realizar um processamento deste nível fazia com que a solução demonstrasse sinais de travamento à medida que o modelo projetado crescia. Dessa forma, foi pensada em uma análise matemática para concluir essa tarefa com menor esforço computacional possível. Assim surgiu a classe *ConnectionLine* que implementa a definição matemática de uma reta. Para isso foi necessário utilizar a equação geral da reta mostrada na equação 5.1.

$$ax + by + c = 0 \tag{5.1}$$

Para encontrar os valores de a, b e c à partir de dois pontos distintos da reta, tomou-se como base o seguinte cálculo matemático:

$$y - y_0 = m(x - x_0) \tag{5.2}$$

Onde

$$m = \frac{y_f - y_0}{x_f - x_0} \tag{5.3}$$

Substituindo a equação 5.3 em 5.2:

$$\begin{aligned} y - y_0 &= \frac{y_f - y_0}{x_f - x_0}(x - x_0) \\ (x_f - x_0)(y - y_0) &= (y_f - y_0)(x - x_0) \\ (x_f - x_0)y - (x_f - x_0)y_0 &= (y_f - y_0)x - (y_f - y_0)x_0 \\ (x_f - x_0)y + (y_0 - y_f)x + (y_f - y_0)x_0 - (x_f - x_0)y_0 &= 0 \end{aligned} \tag{5.4}$$

Comparando o resultado obtido na equação 5.4 com a equação geral da reta em 5.1, podemos perceber que:

$$a = (x_f - x_0),\ b = (y_0 - y_f),\ c = (y_f - y_0)x_0 - (x_f - x_0)y_0 \tag{5.5}$$

As linhas de conexão são definidas logo após a inserção de um novo ponto ao *array* de pontos de cada conexão. Os atributos "a", "b"e "c"de cada reta são definidos de acordo com a equação 5.5. A partir disso, é possível fazer comparações utilizando a função *intersect* de *ConnectionLine* conforme visto na figura 5.3. Esta função verifica se a linha analisada intersecta uma outra linha, passada por parâmetro, fazendo o determinante da matriz formada pelas suas variáveis "a"e "b", de acordo com a equação 5.6.

$$M = \begin{pmatrix} a_1 & b_1 \\ a_2 & b_2 \end{pmatrix} \tag{5.6}$$

Caso o valor do determinante seja 0, não existe colisão entre as duas retas e o valor nulo é retornado. Caso seja diferente de 0, a colisão existe, e é preciso calcular o ponto de intersecção. Para isso, a partir da fórmula geral da equação da reta, consideremos a equação 5.7.

$$\begin{aligned} ax + by + c &= 0 \\ ax + by &= -c \end{aligned} \tag{5.7}$$

Logo, devemos resolver o sistema de equações formado pelas equações gerais das duas retas representados pela matriz da equação 5.8

$$\begin{pmatrix} a_1 & b_1 \\ a_2 & b_2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} -c_1 \\ -c_2 \end{pmatrix} \tag{5.8}$$

Aplicando a regra de Cramer, podemos chegar a fórmulas que representam as coordenadas do ponto de intersecção entre as duas retas, definidos pelas equações 5.9 e 5.10.

$$x_p = \frac{(-c_1 b_2) - (-c_2 b_1)}{|M|} \tag{5.9}$$

$$y_p = \frac{(-c_2 a_1) - (-c_1 a_2)}{|M|} \tag{5.10}$$

Após obter os pontos de intersecção entre as duas retas, resta saber se elas pertencem às linhas desenhadas, através da função *test*, também da classe *ConnectionLine*, que verificará se as coordenadas do ponto encontrado está entre os valores de (x0, y0) e (xf, yf), aplicando, por fim à função geral da reta analisada. Caso o valor seja 0, o ponto encontrado é uma solução para o sistema linear representado pela equação 5.8 e pode ser considerado como um caso de colisão entre conexões distintas.

Sendo assim, a função *draw* poderá, para cada reta da nova conexão, percorrer todas as retas de todas as conexões já existentes na área de desenho, e chamar a função *intersect* para verificar se um ponto de intersecção é encontrado. Em caso positivo, deve ser verificado no contexto do AutoTF se a intersecção já havia sido processada para que seja registrada na lista de intersecções e a imagem do *bypass* seja adicionada no ponto de colisão entre as conexões.

Um retângulo branco deve ser preenchido antes do posicionamento da imagem do *bypass* para apagar qualquer resíduo das linhas já desenhadas. A partir disso pode ser escolhida a imagem vertical ou horizontal do *bypass* que será alocada. Foi definido que a conexão que for criada por último deve ficar abaixo da conexão já existente. Portanto, caso o atributo "a"da linha analisada possua valor nulo, indicando que a reta colidente é horizontal, um *bypass* vertical será adicionado. O contrário vale para quando o valor de "b"possuir valor nulo, indicando que se trata de uma reta vertical.

5.1.5 Interface Gráfica

Uma vez concluído o processo de elaboração das classes que participariam do processo de construção do desenho do modelo físico, havia chegado o momento de se pensar em como seria a interface gráfica, apresentando ferramentas para desenho e manipulação dos objetos, fonte principal de comunicação com o usuário no processo de modelagem. Funcionalidades como a criação, rotação, deslocamento e remoção de objetos, assim como a criação de conexões permitindo escolher quais objetos estariam conectados, ou até mesmo recursos mais genéricos como refazer ou desfazer uma ação, salvar e carregar o trabalho implementado, limpar toda a área de modelagem e executar o comando para se iniciar os cálculos do modelo desenhado, deveriam estar disponíveis na página inicial da solução deste trabalho.

Sendo assim, foram projetados a barra de apresentação com as funções gerais do AutoTF no topo da tela, o menu de ferramentas e paleta de componentes à esquerda, com opções para criar e manipular os elementos do sistema físico. Um rodapé de identificação do projeto que está sendo construído foi posicionado na parte inferior da janela, especificando o propósito deste trabalho. A figura 5.6 nos mostra a configuração final da página

inicial do AutoTF com todos os elementos distribuídos em suas devidas posições.

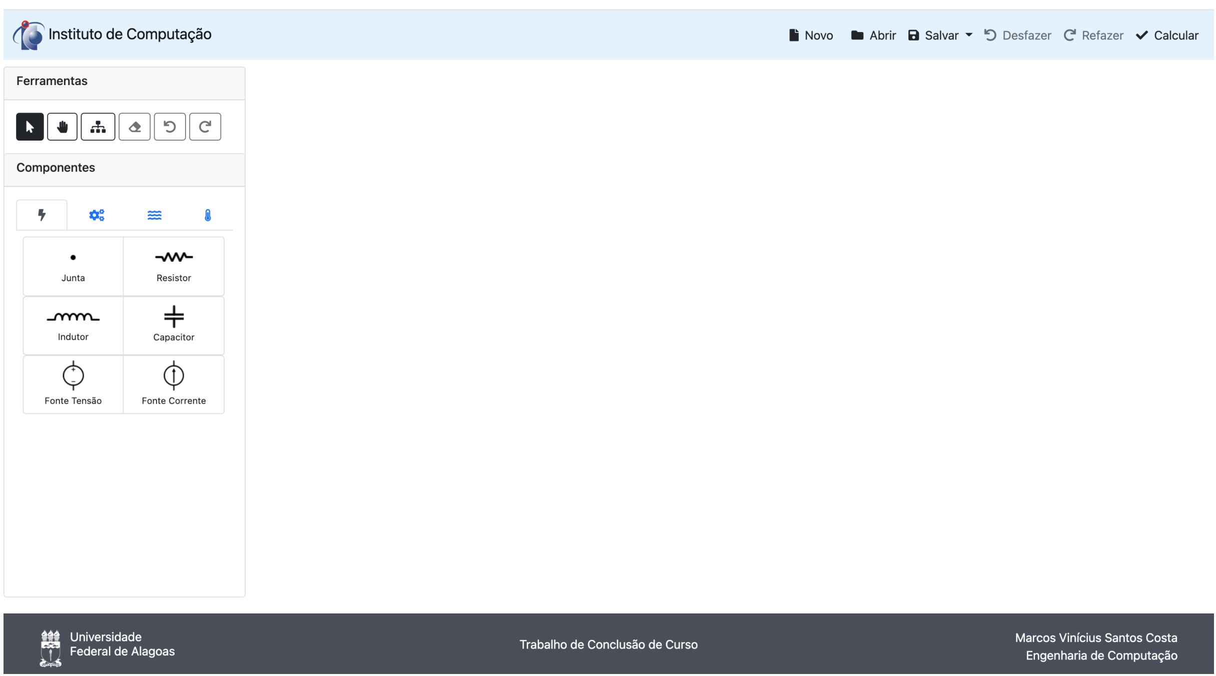


Figura 5.6: Tela inicial da aplicação.

A interface foi construída utilizando o framework *Bootstrap 5*, por proporcionar maior agilidade de desenvolvimento do conteúdo da página, evitando que seja gasto muito tempo na construção da interface gráfica, trazendo também estilos e recursos que já estão familiarizados por boa parte dos usuários que acessam a Internet. Alguns ícones também foram utilizados para melhorar a comunicação visual com o projetista, dando mais clareza a respeito da utilidade de cada recurso. Esses ícones foram extraídos da versão gratuita do *FontAwesome* V5.15.

A página de prototipagem do projeto contará com 3 diferentes modos de atuação: Modo de seleção, modo de deslocamento da área de desenho e modo de construção de conexões. Todas as ações realizadas no canvas acarretarão em comportamentos, de acordo com o modo de atuação selecionado.

Todos os comportamentos realizados por cada um dos recursos mencionados são orquestrados por um arquivo Javascript que chamamos de "contexto", já citado algumas vezes nesse capítulo. É nele que estarão implementadas as funções de construção do desenho, executadas após a modificação na área de desenho, ou resposta à eventos provocados pelo usuário. Estarão também declaradas as variáveis globais do projeto, nesta fase de modelagem do sistema físico, que armazenam dados que de alguma forma controlam o funcionamento geral da aplicação, bem como a existência de elementos no espaço de desenho. A tabela 5.8 nos mostra uma descrição das variáveis de contexto.

Atributo	Descrição
currentState	Objeto de estado atual do sistema, utilizado pelas funções "Desfazer"e "Refazer"
componentModalOpen	*Flag* indicadora de tela de edição de componentes aberta
components	Lista de componentes existentes no desenho
connections	Lista de conexões existentes no desenho
draggingComponent	*Flag* indicadora de operação de deslocamento de componente
draggingWorkspace	*Flag* indicadora de operação de deslocamento da área de desenho
firstSelectedConnector	Primeiro conector selecionado para criação de uma nova conexão
itemSelected	Objeto (componente ou junta) selecionado
newComponent	Componente temporário para exibição de "fantasma"ao posicionar um novo componente na tela de desenho
pointCollisions	Pontos de interssecção entre conexões já tratados pelo sistema
toolSelected	Modo de operação da tela de desenhos: 1- Modo seleção, 2- Modo deslocamento da área de desenho, 3- Modo criação de conexões

Tabela 5.8: Atributos de contexto da etapa de construção do desenho

Menu lateral

O menu lateral (figura 5.7) foi projetado para fornecer ao usuário ferramentas de construção de sistemas. É nele que estarão disponibilizados os recursos de manipulação dos objetos e deslocamento de toda a área de desenho, em sua parte superior; e a paleta de componentes, separados em abas de acordo com a sua natureza (Elétrico, Físico, Térmico e Fluídico), em sua parte inferior. Para a entrega deste trabalho, apenas a aba "Elétrico"está preenchida, ficando as outras como trabalho posterior, bem como a adição de novos componentes elétricos.

Sua parte superior pode ser dividida em dois grupos diferentes: Os três primeiros botões, da esquerda para a direita são botões de alteração de modo de execução da página. Os três últimos são destinados para ações realizadas nos elementos de desenho. O comportamento apresentado após o clique de cada um deles alteram de alguma forma variáveis de contexto e/ou específicas de um objeto selecionado.

- **Modo de operação: Seleção de objetos -** Neste modo de operação será possível selecionar qualquer objeto existente na tela de desenho, bem como alterar seu posicionamento. Este é o modo de operação padrão do AutoTF. Portanto seu uso também é designado para a criação de novos componentes no sistema físico.
  Ao clicar neste botão, um valor nulo é atribuído à variável de contexto *newComponent* limpando qualquer novo componente ainda não posicionado na tela. A variável

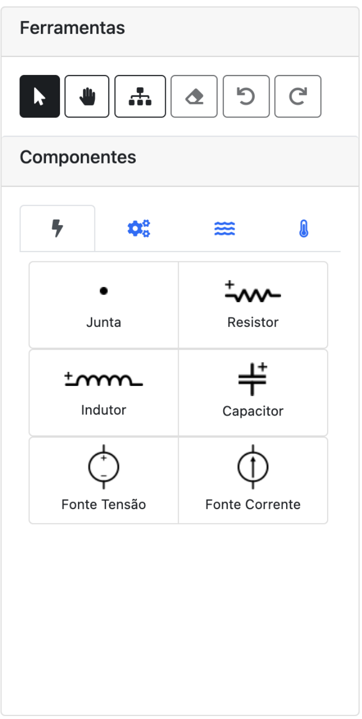


Figura 5.7: Menu lateral da aplicação.

toolSelected passa a assumir o valor 1, desativando qualquer outro botão de escolha de modo de operação que esteja marcado.

- **Modo de operação: Deslocamento da área de trabalho -** Este modo é destinado para ampliar a área de desenho disponível para modelagem de projetos. Ao clicar neste botão a variável de contexto *toolSelected* assume o valor 2, o cursor passa a ter a forma de uma mão aberta, informando que ao clicar e arrastar no canvas, todos os componentes desenhados serão deslocados de acordo com a posição do mouse.

- **Modo de operação: Criação de conexões -** A criação de conexões foi amplamente comentada na seção 5.1.3. Seu ponto de partida acontece ao clicar no terceiro botão da barra de ferramentas. O sistema altera o valor da variável *toolSelected* para 3, e então passa a verificar se o mouse passa sobre algum conector.

- **Excluir elemento -** Este botão estará ativo apenas se existir algum elemento armazenado na variável *itemSelected*. Sua implementação incia-se na verificação do tipo de elemento selecionado. Caso seja um componente ou uma junta, ele é removido da lista de componentes e todas as conexões ligadas a ele são removidas da lista de conexões do contexto. Caso o elemento seja uma conexão, o processo consiste apenas em sua remoção da lista de conexões.

- **Rotacionar componente para a esquerda -** Este botão estará ativo apenas

se existir algum componente armazenado na variável *itemSelected*. Sua execução consiste na chamada ao método *rotateLeft* do objeto selecionado.

- **Rotacionar componente para a direita -** Este botão estará ativo apenas se existir algum componente armazenado na variável *itemSelected*. Sua execução consiste na chamada ao método *rotateRight* do objeto selecionado.

Ao selecionar um componente, o modo de operação será alterado para seleção de objetos (*toolSelected = 1*), qualquer outro botão de componente ativo será desmarcado e a variável de contexto *newComponent* será preenchida com uma instância do elemento escolhido. Sua representação gráfica passará a existir na área de desenho, e sua posição será alterada conforme movimento do mouse, enquanto não for posicionado em algum lugar do projeto.

Eventos do Canvas

De acordo com o que já foi apresentado, pudemos perceber que o *canvas* é uma ferramenta bastante adequada para projetar imagens personalizadas em um ambiente web. Mas seu leque de utilidades se expande quando adicionamos eventos à sua implementação, fazendo com que ele responda às interações do usuário modificando o espaço desenhado. Para atender aos propósitos deste trabalho na etapa de modelagem, foi necessário configurar alguns eventos, como descrito à seguir:

- Clique simples:
  - Caso o modo de seleção esteja ativo e a variável *newComponent* esteja armazenando um novo componente, sua posição deve ser fixada, adicionando o novo elemento à lista de componentes do contexto, e gerando outro componente para armazenar em *newComponent*.
  - Caso o modo de seleção esteja ativo e a variável *newComponent* assuma valor nulo, a posição do ponteiro do mouse é capturada para que se faça uma checagem em cada elemento existente se as coordenadas estão dentro da região delimitada por eles. Em caso positivo, sua variável *selected* é alterada para *true* e o contexto é redesenhado para que o componente selecionado seja destacado.
  - Se o modo de criação de conexões estiver ativo, é verificado se algum conector ou junta foi selecionado, utilizando a mesma lógica aplicada aos componentes. Em caso positivo, precisa-se saber se algum outro conector já havia sido selecionado previamente, significando que o elemento selecionado é o destino da conexão a ser criada. Uma vez identificado que o conector selecionado é o primeiro, o objeto é armazenado na variável de contexto *firstSelectedConnector*, fazendo com que uma linha reta seja desenhada à partir do conector até

o ponteiro do mouse, informando que é necessária uma segunda seleção para completar o processo. No segundo clique, é verificado que *firstSelectedConnector* não mais possui valor nulo, uma nova conexão é criada com origem e destino já estabelecidas (caso os elementos de origem e destino sejam diferentes), e a nova instância é armazenada na lista de conexões do contexto, concretizando a criação da entidade. Se os elementos envolvidos pertencerem à classe *Connector*, é realizada uma varredura na lista de conexões para remover quaisquer outras conexões relacionadas à eles. Ao fim do processo, *firstSelectedConnector* volta a ter valor nulo, e o ciclo se reinicia.

- Clique duplo: A única necessidade existente para o uso do duplo clique na área de desenho é abrir o painel de edição de componentes. Para isso, o sistema deve estar no modo de operação de seleção de objetos. É identificado então se as coordenadas do mouse estão sobrepostas a algum componente. Em caso positivo, o título da janela de edição de componentes e seus *inputs* são preenchidos com as informações atuais do componente selecionado. Uma lista de opções é carregada no *select* ao lado do campo valor com todos os múltiplos da unidade de medida relacionada ao componente. O evento para o botão salvar é implementado, fazendo com que ao ser clicado, as novas informações sejam atualizadas nas variáveis do objeto destacado. A tela de edição de componentes (figura 5.8 para edição de um resistor) é então apresentada no navegador com todas as informações preenchidas.

Figura 5.8: Janela de edição de componente

- Pressionar botão principal do mouse:
  - Caso o modo de seleçao de objetos esteja ativo e não exista nenhum componente sendo criado (variável *newComponent* nula), é verificado se o mouse está

sobre algum componente ou junta. A seleção é então realizada, e as coordenadas atuais do clique e do componente são armazenadas em variáveis globais para que seja possível realizar o deslocamento do objeto baseado nas próximas posições do mouse. A *flag* de contexto *draggingComponent* é marcada com valor *true* e um novo evento é cadastrado, para que ao liberar o botão do mouse, *draggingComponent* volte a ter valor *false* e um novo estado de contexto seja armazenado, caso as posições iniciais e finais do elemento sejam diferentes. Após o disparo deste segundo evento, o mesmo é removido para que não haja conflitos com novas ações do usuário.

- Caso o modo de deslocamento da área de desenho esteja ativo, o comportamento é análogo ao caso anterior. Mas dessa vez, são armazenadas as posições de todos os elementos existentes no canvas, para que todos sejam deslocados ao mesmo tempo. A variável de contexto que controla esse comportamento é a *draggingWorkspace*. O ícone do cursor do mouse também passa a assumir a imagem de uma mão fechada, indicando que os componentes estão em processo de deslocamento. Ao liberar o botão do mouse, todas as ações são desfeitas, mantendo apenas as novas posições dos elementos em tela.

- Mover mouse:
  - Caso a *flag* do contexto *draggingWorkspace* esteja marcada, a lista de componentes é percorrida e a posição de cada um deles é atualizada conforme a equação 5.11, onde x_0 e x_n representam respectivamente as antigas e as novas coordenadas do elemento e $mouseX_0$ e $mouseX_f$ as coordenadas do mouse no momento do clique no momento atual, enquanto ocorre o movimento.

$$x_n = x_0 + (mouseX_f - mouseX_0) \tag{5.11}$$

    Em suma, podemos definir a equação como o incremento de variação das coordenadas do mouse às coordenadas iniciais do elemento.

  - O mesmo acontece para quando a *flag* do contexto *draggingComponent* estiver marcada. A única diferença é que ao invés de mover todos os componentes, apenas o elemento contido na variável *itemSelected* é reposicionado.
  - Caso nenhum dos itens anteriores sejam válidos e o sistema esteja em modo de criação de conexões, mover o mouse vai fazer com que sejam verificados se o ponteiro está sobrepondo alguma junta ou conector, fazendo a devida marcação através de suas funções *onMouseOver* descritas na seção 5.1.1. Caso algum dos dois tipos de elementos esteja sendo sobreposto, o ponteiro do mouse terá seu ícone alterado representar possível clique. Ao sair da área do conector o ícone é redefinido para a seta. Caso *firstSelectedConnector* assuma valor diferente de

nulo, uma reta deve ser desenhada tendo como ponto de partida as coordenadas do primeiro conector selecionado e destino a posição atual do mouse, mostrando que o sistema está aguardando a seleção de um segundo conector.

– A não ocorrência de nenhuma das situações anteriores fará que o sistema verifique se existe algum componente na variável *newComponent* indicando possível novo elemento no projeto. A posição desse elemento será atualizada conforme coordenadas do mouse enquanto um evento de clique não for disparado.

Menu superior

O menu superior (figura 5.9) foi projetado para auxiliar o usuário com funções gerais do sistema AutoTF, permitindo a realização de tarefas comumente observadas em *softwares* de edição. As ações dos botões presentes nesse menu impactam o trabalho desenvolvido pelo projetista como um todo. À partir dessas funções é possível salvar o trabalho atual, carregar um projeto salvo anteriormente, desfazer ou refazer alterações na área de desenho ou até mesmo limpar toda a área de trabalho. Também é a partir desse menu que são obtidos os resultados da modelagem do sistema físico desenhado.

Figura 5.9: Ferramentas do menu superior da aplicação.

Descreveremos abaixo cada uma das opções presentes na barra superior deste trabalho:

- **Novo Projeto -** Esta opção esvazia a lista de componentes e conexões do contexto, pedindo apenas uma confirmação do usuário caso exista algum componente projetado em tela.

- **Salvar -** O AutoTF permite que o projeto implementado seja salvo em dois formatos diferentes: PNG, onde uma cópia da tela do canvas é extraida para registro apenas da imagem visível na tela, ou JSON armazenando todos os objetos presentes no contexto atual em arquivo de texto. A função "Abrir"estará disponível apenas para projetos salvos no formato JSON. A conversão dos objetos para texto deve ser personalizada, para que sejam evitados *loops* infinitos por conta da dependência mútua entre os objetos. Para isso, cada classe envolvida deve ter implementada uma função denominada *toJsonObject* retornando uma *string* do objeto no formato *json*. As listas de conexões e componentes do contexto são percorridas, adicionando no arquivo os resultados desta função em cada um dos elementos.

- **Abrir -** Ao clicar nesta função, será aberta uma caixa de seleção de arquivos, para que possa ser escolhido o arquivo no formato JSON gerado pelo AutoTF em

algum momento anterior. O processo que acontece em seguida é o inverso do que é realizado na função "Salvar". O texto do arquivo é transformado em uma lista de objetos Javascript separados entre componentes e conexões. As listas de elementos do contexto serão preenchidas com os novos objetos gerados à partir das variáveis lidas, em cada elemento contido no arquivo.

- **Desfazer/Refazer -** Para implementar as opções de desafazimento e refazimento de alguma ação, é necessário que objetos de estado sejam armazenados a cada operação realizada pelo usuário em uma estrutura de dados conhecida como "listas duplamente encadeadas". Caso não existisse a implementação da função "Refazer", o armazenamento em pilhas seria o suficiente. A função de gerar elementos de texto com os componentes existentes no contexto é aproveitada, e cada estado armazena uma "cópia"das listas de elementos. A existência de vários arquivos de *backup* podem exemplificar esse modelo. Esses estados estão interligados por variáveis *next* e *previous*, indicando os estados anteriores e, em caso de um desfazimento, o próximo estado a ser executado. Dessa forma, ao realizar uma operação de desfazer, a variável de contexto *currentState* passa a apontar para o estado contido na sua variável *previous*, caso exista. O mesmo acontece com a função Refazer, onde *currentState* passa assumir o endereço do estado contido em *next*, caso não seja nulo. A existência de estados não nulos nas variáveis *previous* e *next* indicarão se os botões de função "Desfazer"e "Refazer", respectivamente, estarão ativos e disponíveis para clique.

- **Ver Resultados -** Esta é a opção mais importante do menu superior. É a partir dela que o sistema desenhado é validado, para que a tela de resultados parciais seja chamada. Essa opção representa a continuidade do fluxo do AutoTF e foi projetada no menu superior para que seja apresentada em destaque, mostrando ao usuário o que precisa ser feito após finalização do desenho do modelo físico.

5.1.6 Validação do sistema físico

O processo de validação do sistema projetado se inicia verificando se a lista de componentes no contexto possui algum elemento. Uma mensagem de alerta ao usuário é mostrada, conforme a figura 5.10, caso nenhum componente tenha sido desenhado.

Uma vez constatado que existem componentes desenhados, deve ser realizada uma análise em todos os objetos que implementam a classe *Connectable* para verificar se estão associados à alguma conexão. Para os objetos de juntas, devem ser encontradas pelo menos duas conexões associadas a eles para que o sistema seja válido. A partir disto, foi preferível transformar a lista de conexões e a lista de componentes em uma estrutura única em formato de grafo, representado através de listas de adjacências para que as conexões entre juntas fossem eliminadas e uma verificação do conectividade fosse realizada, evitando

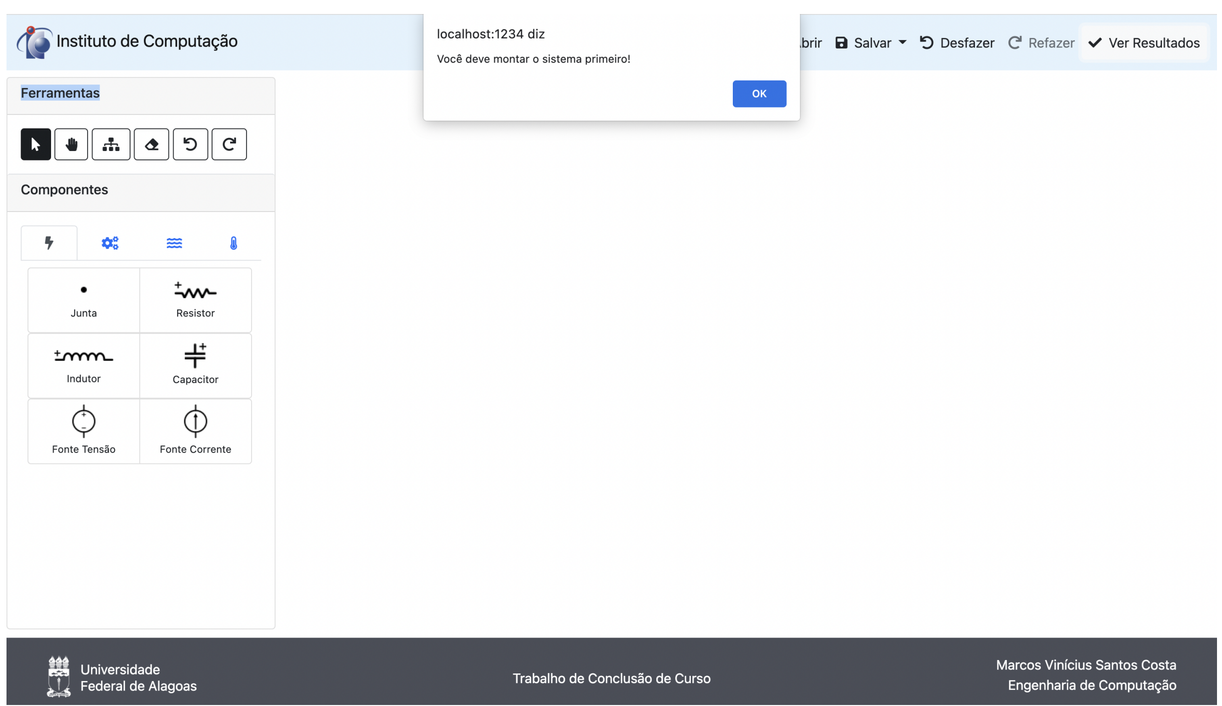


Figura 5.10: Mensagem de erro de validação de sistema.

assim a presença de malhas isoladas.

Transformação do sistema em grafo

Este processo consiste em percorrer a lista de conexões armazenada no contexto e para cada iteração definir estruturas de dados personalizadas de acordo com os componentes conectados pela conexão analisada. A nova estrutura de dados denominada *GraphElement* contará com as variáveis descritas na tabela 5.9 e novas instâncias serão geradas para cada um dos componentes e juntas existentes no protótipo do usuário.

Atributo	Descrição
uuid	Token único de identificação do elemento. Será o mesmo atribuído para o componente ou junta
connectorId	Identificador do conector utilizado na conexão (A ou B). Para junta, o valor "*Joint*"é assumido
object	Instância do componente ou junta utilizado pelo contexto
parentConnectorId	Identificador do conector do elemento conectado (A ou B). Quando conectado a uma junta, o valor "*Joint*"é assumido
parentObject	Instância do componente ou junta ao qual o componente observado está conectado

Tabela 5.9: Atributos da classe *GraphElement*

Os objetos gerados são adicionados à lista de adjacências do elemento oposto, identificadas pelo seu *uuid*. As ligações em que ambos os componentes conectados pertencem

à classe *Joint*, são adicionadas a uma lista para que essas conexões sejam removidas.

O processo de remoção de conexões entre juntas deve ser realizado para simplificar o sistema modelado. Consiste em atribuir todas as conexões relacionadas à uma junta B à junta A, a qual está conectada. Após esse procedimento, a junta B pode ser excluída sem que o sistema físico proposto seja alterado. Para isso, é necessário percorrer a lista de conexões *Joint-Joint*. Por convenção, a primeira junta permanecerá no grafo, enquanto segunda entrará em processo de exclusão. Dessa forma, é necessário remover a instância de *GraphElement* da junta B da lista de adjacências de cada um dos elementos ao qual ela está associado, adicionando um elemento equivalente à junta A, caso já não exista a conexão entre eles. Por fim, é removido do grafo toda a lista de adjacências identificadas pelo atributo *uuid* da junta B.

Verificação de conectividade do grafo

Para completar a validação do sistema proposto pelo usuário, basta que seja constatado que o desenho não possui malhas isoladas. Sendo assim, o grafo gerado na seção anterior foi submetido a uma análise para armazenar o nível de alcance de cada um de seus nós.

O processo consiste na execução do algoritmo de busca em profundidade, a partir de um nó aleatório do grafo, observando-se quais nós foram visitados em apenas uma execução desta rotina. Caso algum nó não tenha sido visitado após a execução do algoritmo no grafo, o sistema é dito inválido e uma mensagem é informada ao usuário. Caso contrário, o painel de resultados parciais é carregado e a operação segue o seu fluxo para criação do grafo de ligação e do diagrama de fluxo de sinais.

5.2 Painel de Resultados Parciais

Após encerrada a etapa de prototipagem do modelo físico, é preciso gerar os elementos intermediários, necessários para a obtenção do grafo de ligação. O painel de resultados parciais foi criado com o propósito de externalizar esses resultados para o usuário, para que ele possua meios de verificar como que o AutoTF chegou ao resultado final, tornando possível uma melhor compreensão dos dados, até mesmo como meio de constatar a existência de algum erro. O painel é dividido em 9 abas, sendo uma para o grafo de ligação, 7 abas para as matrizes de interconectividade ($\mathbf{A_1}$, $\mathbf{A_2}$, $\mathbf{A_3}$, $\mathbf{A_4}$, $\mathbf{A_5}$, $\mathbf{A_{5N}}$ e $\mathbf{A}$), e a última para exibição do DFS. As abas estão ordenadas de acordo com a ordem de execução de cada procedimento.

5.2.1 Interface Gráfica

A interface gráfica presente nas abas estão divididas basicamente em dois grupos. O desenho dos grafos e a apresentação das tabelas. Para gerar a representação do grafo

também foi utilizado o canvas. Mas para essa etapa, a configuração desejada é bem mais básica, uma vez que não é preciso pensar em recursos como elaboração de conexões complexas ou controle de colisão. As conexões entre os vértices são realizadas utilizando estruturas denominadas "ramos"que contém o valor indicativo de peso da aresta. Os vértices são representados como circunferências e os ramos como quadrados com linha tracejada, ambos com letreiro central contendo o id do vértice ou o identificador do tipo de ramo. Todas as linhas que conectam vértices aos ramos podem ser desenhadas a partir de uma única reta. A única diferença é que como o grafo de ligação e o diagrama de fluxo de sinais são dígrafos, é preciso desenhar uma seta para que possa ser indicada a direção da aresta. Para isso devemos calcular o ângulo de inclinação da reta, que é definido por θ, na equação 5.12.

$$\theta = \arctan \frac{(y_f - y_0)}{(x_f - x_0)} \tag{5.12}$$

Estabelecendo o ângulo de inclinação das retas componentes da seta como 30°, de tamanho 10px, basta utilizarmos as relações trigonométricas no triângulo para calcular os pontos finais para construção das retas, levando em consideração que o ponto de origem é o centro da reta da conexão (x_m, y_m). As coordenadas dos pontos finais para as retas das setas são definidas pela equação 5.13

$$\begin{aligned} x &= x_m - 10\cos{(\theta \pm \frac{\pi}{6})} \\ y &= y_m - 10\sin{(\theta \pm \frac{\pi}{6})} \end{aligned} \tag{5.13}$$

Após modelar o desenho do grafo, basta apenas saber quais interações estão disponíveis ao usuário. Nesta etapa, não é necessário outros comandos além do deslocamento de objetos ou de toda a área do grafo. Dessa forma, alteramos o ícone do cursor do mouse para a imagem da mão aberta enquanto o botão do mouse não estiver pressionado, para que após iniciar um processo de deslocamento seja alternada com o ícone da mão fechada. Toda a lógica utilizada no deslocamento de objetos da etapa de desenho do sistema físico é aplicada nos grafos de resultados.

A apresentação das matrizes acontece de forma mais simples, criando uma *tag* HTML *table* nas abas correspondentes, e utilizando o Javascript para preenche-las dinamicamente à medida que os resultados forem sendo obtidos.

5.2.2 Construção do Grafo de Ligação

A representação do grafo de ligação foi implementada utilizando as mesmas técnicas da construção do desenho do sistema. Dessa forma, apesar de conceitualmente se tratar de um grafo, sua estrutura de dados no AutoTF se assemelha a uma lista de elementos de diferentes tipos que são elementos da estrutura gráfica do grafo de ligação. São eles:

GraphComponent, representando o ramo, *GraphNode*, representando o vértice e *GraphArrow* como a linha que liga um ramo a um vértice. Sua estrutura de dados é definida pela figura 5.11.

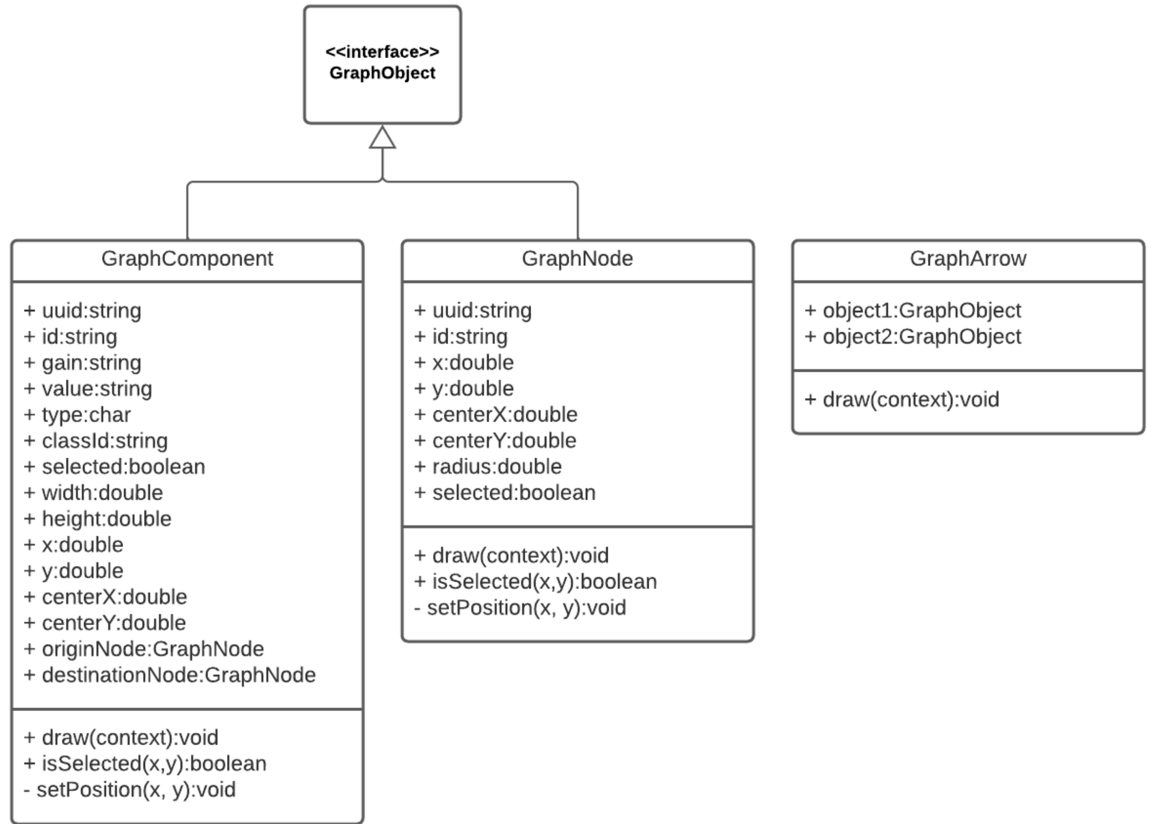


Figura 5.11: Diagrama de classes da estrutura do grafo de ligação.

Os atributos apresentados no diagrama seguem o mesmo padrão de nomenclatura utilizado nos objetos já apresentados até aqui. Portanto, a tabela 5.10 mostrará apenas os novos atributos pertencentes à classe *GraphComponent*, pois terão papel fundamental na execução das próximas etapas.

A criação dos elementos da nova estrutura acontece à partir do grafo do sistema físico, gerado na etapa de validação do projeto do usuário, comentado na seção anterior. Como a identificação dos vértices deve ser definida de forma sequencial, começando a partir do 1, uma variável para armazenar esse valor é inicializada e então, cada uma das listas de adjacência vão sendo percorridas, para que seja realizada uma análise a partir dos dois elementos envolvidos, e assim, definir qual o tipo de estratégia será aplicada.

Caso um dos elementos seja uma junta, esta será transformada em um nó do grafo de ligação, e à partir do componente físico será criado o vértice. Como a direção dos ramos deve ser estabelecida no sentido positivo do fluxo e contrário à quantidade de esforço existente em cada um dos vértices, para definir a direção da ligação entre o vértice e o ramo, é observado qual dos terminais do componente está conectado com a junta.

Atributo	Descrição
gain	Valor simbólico do ganho do componente, utilizado para calcular a função de transferência
value	Valor numérico do componente, já considerando os múltiplos da unidade de medida
type	Tipo de ramo do grafo de ligação: (E: Fonte de esforço, F: Fonte de fluxo ou Z: Impedância)
classId	Identificador da porta do elemento, a partir do elemento original. Pode assumir os valores: (Impedance, FlowFont, EffortFont, ControlledFlowFont, ControlledEffortFont, 1-TransformerPort, 2-TransformerPort, 1-RotatorPort, 2-RotatorPort)

Tabela 5.10: Novos atributos da classe *GraphComponent*

A direção da linha de conexão será do vértice para o ramo se o terminal envolvido for o positivo (*connectorId* = 'B'), e o elemento seja uma fonte de tensão, ou o terminal envolvido for o negativo (*connectorId* = 'A') e o elemento seja uma fonte de corrente ou um dos elementos de impedância. Caso contrário, a conexão será no sentido ramo-vértice.

Se não houverem juntas envolvidas na conexão analisada, serão gerados dois ramos e um vértice entre eles de modo que a direção da linha conectora será definida utilizando a mesma lógica aplicada no parágrafo anterior.

O indicativo de direção para a linha que conecta ramos à vértices acontece por meio das variáveis *object1* e *object2*, de modo que o sentido da seta direcional ocorre sempre do *object1* para o *object2*.

Dessa forma os elementos do grafo de ligação vão sendo gerados e armazenados em um novo contexto, dedicado à esta etapa, possuindo sua própria função *draw* para que sejam desenhados no canvas.

5.2.3 Cálculo das matrizes de interconectividade

A construção das matrizes de interconectividade é uma etapa essencial para obter o diagrama de fluxo de sinais. Para que ela seja realizada, foi necessária a criação de um novo contexto, que será o responsável por assumir o controle do AutoTF até o momento apresentação da DFS em tela. Para que ele seja inicializado, são necessárias a lista de elementos do grafo de ligação, um grafo auxiliar representando o grafo de ligação de maneira não direcionada e uma lista dos componentes conexos deste grafo auxiliar.

Para atender os dois últimos requisitos, foi implementada uma função semelhante ao processo de transformação do desenho do sistema físico em grafo, que percorre a lista de elementos do grafo de ligação e cria uma estrutura, identificada pelo *uuid*, para cada objeto do tipo vértice *GraphNode* ou ramo *GraphComponent*. A estrutura chamada *AuxGraphElement* possui os seguintes atributos, apresentados na tabela 5.11.

Atributo	Descrição
adjacencyList	Lista de adjacências indicando a quais elementos o objeto está conectado
object	Instância do elemento do grafo de ligação que gerou esta estrutura
type	Tipo da instância do elemento que gerou esta estrutura: (*"branch"* para ramos, *"node"* para vértices)

Tabela 5.11: Atributos da classe *AuxGraphElement*

Os objetos do tipo *GraphArrow* são utilizados para preencher as lista de adjacências, considerando que o grafo deve ser não direcionado. Assim, para cada conexões, são realizadas duas inserções nas listas de adjacência, uma de cada elemento.

Para encontrar os componentes conexos do grafo auxiliar, basta rodar o algoritmo de busca em profundidade marcando os objetos encontrados, e enquanto todos não forem marcados, uma nova execução do algoritmo é realizada, separando em diferentes listas quem foi visitado em cada execução do algoritmo.

Uma vez atendido todos os pré requisitos, é possível inicializar as variáveis do DFS. O primeiro passo é separar em listas distintas os diferentes tipos de elementos do grafo de ligação. As quantidades de ramos e de vértices serão utilizadas para calcular as matrizes. Em seguida, devem ser definidas as variáveis de entrada e saída do DFS à partir da lista de ramos. Para este trabalho, foi preferido ordenar os ramos de modo que os primeiros elementos seriam as fontes de tensão, em seguida as fontes de corrente, fontes controladas (não implementadas) e por fim as impedâncias (Os elementos de duas portas estariam após as impedâncias quando forem implementados), para que se tenha uma melhor visualização no momento da apresentação dos resultados.

Sendo assim, para cada ramo foram geradas duas estruturas *DFSVariable* e *DFSVariableConnections* representadas pela figura 5.12

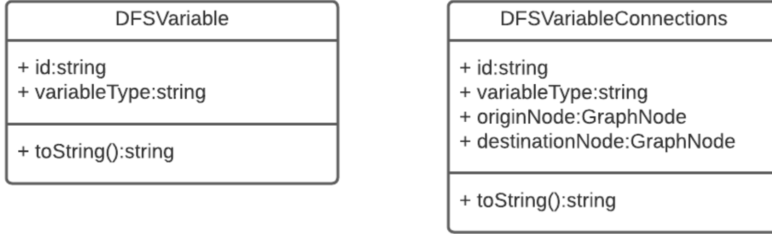


Figura 5.12: Diagrama de classes das variáveis do DFS.

O atributo *variableType* indica se o variável é de esforço ou de fluxo podendo assumir os valores *"effort"* ou *"flow"*. O id é preenchido de acordo com o id do componente do grafo de ligação servindo para identificar a qual elemento a variável está se referindo.

Dessa forma, para cada elemento é gerado cada uma das estruturas apresentadas na

figura 5.12, sendo uma para cada tipo de variável. Os objetos de *DFSVariable* do tipo *effort* associadas a todos os elementos de fonte de tensão e as variáveis de fluxo associadas à fontes de corrente são adicionadas à lista de variáveis de entrada. As demais formam a lista das variáveis de saída. Por fim, são criadas duas listas gerais, uma de elementos *DFSVariable* e outra de *DFSVariableConnections* ordenadas de acordo com a ordem dos ramos, sendo que a primeira variável de fluxo estará localizada após a última variável de esforço.

As variáveis da DFS agora estão criadas e todo o cenário está pronto para se iniciar a geração das matrizes de interconexão dos elementos físicos. Todas as matrizes terão o mesmo número de colunas, onde cada uma delas vai representar uma das variáveis do DFS.

Matriz A_1

A primeira é chamada de matriz de impedâncias. Ela é gerada à partir dos valores dos ganhos de cada uma das impedâncias existentes no projeto, de acordo com a posição das variáveis de esforço e fluxo de cada elemento analisado.

Cada uma de suas linhas vai representar um componente diferente. Para cada componente é atribuído o seu valor de impedância na coluna que corresponde à sua variável de fluxo e o valor -1 na coluna que corresponde à sua variável de esforço.

Matriz A_2

A segunda matriz é chamada de matriz de fontes controladas. Cada uma de suas linhas serão relacionadas à uma fonte controlada existente no sistema. Como esses componentes não foram implementados na versão proposta para este trabalho, essa matriz ficará sempre vazia. Mesmo assim, o algoritmo necessário para sua criação foi desenvolvido, considerando trabalhos futuros que possam ser adicionados em versões futuras do AutoTF.

A forma de preenchimento da matriz $\mathbf{A_2}$ segue o mesmo padrão que a matriz A1. No entanto, ao invés de adicionar o valor de impedância do elemento na coluna da variável de fluxo do componente, é verificado inicialmente se a fonte controlada se trata de uma fonte de esforço ou de fluxo. Assim, a coluna da matriz referente à variável principal do componente analisado é preenchida com o valor de sua variável de controle. A coluna referente à variável restante assume o valor -1.

Matriz A_3

A matriz $\mathbf{A_3}$ é a matriz de transformação e giração. Assim como a matriz $\mathbf{A_2}$, ela se apresentará sempre vazia uma vez que transformadores e giradores não estão disponíveis na solução deste trabalho. No entanto, o código para geração dessa matriz foi escrito considerando uma futura implementação desses elementos.

O número de linhas da matriz $\mathbf{A_3}$ corresponde ao dobro da quantidade de elementos de duas portas. A forma como cada linha será preenchida depende do tipo de componente que está sendo analisado, de acordo com suas características físicas e o impacto em que seu comportamento trás para o sistema físico.

O preenchimento da matriz é representado pelas tabelas 5.12 e 5.13, onde G equivale ao valor do ganho ramo da primeira porta de cada componente.

E(P1)	**E(P2)**	**F(P1)**	**F(P2)**
	-1	G	
1			G

Tabela 5.12: Preenchimento da matriz A3 para o elemento Girador

E(P1)	**E(P2)**	**F(P1)**	**F(P2)**
-1	G		
		G	1

Tabela 5.13: Preenchimento da matriz A3 para o elemento Transformador

Matriz A_4

A matriz $\mathbf{A_4}$ é chamada de matriz de compatibilidade de esforço. É nela em que as malhas do sistema projetado serão analisados, baseando-se na lei das malhas de Kirchhoff. Foi utilizado o grafo auxiliar formado à partir do grafo de ligação identificando os ciclos existentes nele para que sejam evidenciadas as malhas existentes no circuito desenhado pelo usuário.

Para isso à partir do conceito de árvore geradora do grafo na no grafo auxiliar montado à partir do grafo de ligação, retirando algumas arestas (juntamente com o ramo que a quantifica). Assim, a adição de qualquer uma das arestas removidas à arvore geradora, resultaria em um ciclo.

Para encontrar a árvore geradora foi escolhido um algoritmo bem conhecido da literatura: o algoritmo de Kruskal, que apresenta vantagens em relação aos outros utilizados com o mesmo propósito, com a condição de que os pesos das arestas do grafo a ser processado estejam ordenados. Como na atual situação o valor de cada aresta (que representamos pelo ramo) não influencia no resultado final que procuramos, assumimos que todas possuem o mesmo valor, fazendo com que fosse possível tirar proveito do algoritmo em sua forma de execução ótima.

Após obter a árvore geradora do grafo auxiliar e a lista de ramos removidos, cada um dos ramos foram sendo adicionados individualmente à árvore para que fosse executado um algoritmo que implementa de forma modificada a busca em profundidade no grafo

resultante, iniciando à partir do vértice origem do ramo removido. O algoritmo de busca encerraria sua rotina ao encontrar mais uma vez o vértice inicial, indicando que o ciclo foi encontrado e retornando a lista de vértices e a lista de ramos que ele precisou visitar.

Como o grafo auxiliar não é um dígrafo, duas novas listas de objetos da classe *DFS-VariableConnections* são criadas para que possa ser identificado o sentido das conexões. A primeira lista vai ser preenchida com as variáveis de esforço de cada elemento, de modo que o vértice de origem e destino estejam ordenados de acordo com a ordem de busca aplicada ao grafo auxiliar. A segunda lista será equivalente à primeira, mas com os atributos *originNode* e *destinationNode* com valores invertidos.

Dessa forma é possível procurar cada elemento de cada uma das duas listas formadas na lista de *DFSVariableConnections* do contexto criada no início desta seção. Caso a variável da primeira lista seja encontrada na lista do contexto da aplicação, sua coluna será preenchida com o valor 1. Caso contrário, a variável da segunda linha é a que representa o sistema original, e o valor -1 vai ser adicionado à coluna que o representa na matriz.

Cada um dos ramos removidos representa um ciclo no sistema, e uma linha será criada na matriz A4 para cada um desses ramos. Logo, podemos afirmar que essa matriz vai possuir quantidade de linhas de acordo com a quantidade de malhas que o sistema projetado possui.

Matriz A_5

A matriz $\mathbf{A_5}$, ou matriz de continuidade de fluxo representa uma generalização da lei dos nós de Kirchhoff, e analisa todos os nós existentes no sistema. Portanto, assim como a matriz $\mathbf{A_4}$ tem mesmo número de linhas que a quantidade de malhas, a matriz A5 vai possuir mesmo número de linhas que a quantidade de nós do sistema físico.

Para calcular seus valores, serão utilizados os componentes conexos do grafo auxiliar encontrados no início desta seção. Na solução atual, apenas um componente conexo será encontrado para qualquer sistema projetado, já que a paleta de componentes não é composta por elementos de duas portas. Os nós de cada componente conexo serão processados, e os ramos adjacentes a eles serão analisados para preenchimento da matriz.

Para cada uma das arestas do sistema, deve ser verificado se o vértice que está em processamento é o vértice origem ou destino associado. Caso o vértice observado seja o nó de origem do ramo, é adicionado o valor 1 à coluna que referencia a variável de fluxo do elemento contido no ramo. Se o vértice observado for o destino do ramo, o valor adicionado à matriz será -1.

A fim de garantir que a matriz de interconectividade final represente um sistema linear com solução definida, isto é, com mesmo número de linhas que a quantidade de saídas do sistema, a matriz $\mathbf{A_5}$ deve ser linearmente independente. Dessa forma, para cada componente conexa, deve ser removida a linha que possuir maior quantidade de

elementos não nulos, formando assim a matriz $\mathbf{A_{5N}}$.

Matriz A

Esta é a matriz resultante das relações de interconexão entre os componentes do sistema físico modelado, que pode ser convertida em equações que representam o comportamento do sistema.

A matriz A é formada a partir das 5 matrizes obtidas nas seções anteriores. A ordem utilizada na construção da matriz A foi: $\mathbf{A_4}$, $\mathbf{A_1}$, $\mathbf{A_2}$, $\mathbf{A_3}$ e $\mathbf{A_{5N}}$, pois como as matrizes $\mathbf{A_1}$, $\mathbf{A_2}$ e $\mathbf{A_3}$ tendem a ter maior quantidade de números nulos, colocar a matriz $\mathbf{A_4}$ nas primeiras linhas reduziria o processamento computacional utilizado pelo algoritmo de busca das variáveis que será apresentado logo a seguir. Dessa forma, ela possui quantidade de colunas igual a quantidade de variáveis existentes na DFS e quantidade de linhas igual ao número variáveis de saída. Resta apenas identificar cada linha a partir de uma analise combinatória para saber qual variável de saída é cada uma está relacionada.

5.2.4 Construção do DFS

Para iniciar o processo de montagem do DFS, um grafo representado por listas de adjacências deve estar inicializado com vértices que correspondem a cada uma das variáveis presentes no sistema, de esforço ou de fluxo, inicialmente sem nenhuma conexão entre elas. Os vértices do DFS são identificados por um valor inteiro, iniciando em 1.

É necessário identificar cada uma das linhas da matriz **A**. A partir dos procedimentos executados nas seções anteriores, sabemos que as linhas possuem pelo menos dois elementos não nulos. Dessa forma, uma linha será composta por um valor da variável que ela representa, mais outros valores correspondentes a outras variáveis. É assim que se formam as equações lineares que relacionam os componentes entre si.

A estratégia é utilizar um algoritmo de busca que utilize uma técnica de *backtracking* para tentar montar uma sequência de variáveis por tentativa e erro, de modo que uma variável já identificada não possa ser reutilizada em qualquer outra linha. O AutoTF conseguirá calcular a função de transferência se for possível atribuir cada uma das variáveis de saída a alguma linha de **A**.

Após encontrar a sequencia de variáveis de saída, é o momento de montar o diagrama de fluxo de sinais utilizando os valores inseridos na matriz. Cada linha da matriz **A**, associada com sua variável deve ser percorrida, e para cada valor pertencentes à outras variáveis é adicionada uma conexão entre o vértice da outra variável com vértice da variável da linha. O valor do peso da conexão recém formada é calculado de acordo com equação 5.14, onde G_o é o ganho da variável relacionada, e G_l é o ganho da variável da linha da matriz.

$$G = -\frac{G_o}{G_l} \tag{5.14}$$

A estrutura de dados utilizada para representar o diagrama de fluxo de sinais em tela é a mesma utilizada para representação do grafo de ligação, alterando apenas os nomes das classes. Enquanto os elementos para desenho grafo de ligação eram *GraphNode*, *GraphComponent* e *GraphArrow*, o DFS dispõe de *DFSNode*, *DFSProcedure* e *DFSArrow* para compor suas ferramentas para desenho no canvas.

O processo de representação gráfica consiste em percorrer os vértices existentes na estrutura do grafo construído, e para cada elemento da lista de adjacências são criados, caso não existam o nó de origem, o nó de destino, uma tramitância com mesmo valor do peso da conexão calculado anteriormente e duas linhas de conexão para ligar o vértice inicial à tramitância e a tramitância ao vértice final.

5.3 Apresentação da Função de Transferência

Uma vez obtido o diagrama de fluxo de sinais que representa o sistema físico projetado, o usuário poderá clicar no botão Função de Transferência, presente na parte inferior direita do painel de resultados parciais. Se o algoritmo de busca tiver obtido êxito ao encontrar as variáveis de saída dentro da matriz **A**, uma nova janela será aberta solicitando ao usuário que informe a variável de entrada e a variável de saída desejada para que seja calculada então a função de transferência. A figura 5.13 mostra a interface gráfica construída para a janela de apresentação dos resultados finais.

Se as variáveis não puderem ser encontradas, um informativo aparecerá para o usuário indicando que não é possível encontrar a solução do sistema.

Caso contrário, após definir esses atributos, um novo contexto é definido, dedicado apenas à tarefa de realizar o cálculo final da função de transferência.

5.3.1 Aplicação da Regra de Mason

Para aplicar a regra de Mason no DFS, é preciso primeiramente calcular o valor de seus atributos. É preciso então percorrer o diagrama para encontrar os caminhos diretos e ciclos envolvendo o vértice inicial e o vértice final.

Para este fim foi utilizado mais uma vez o algoritmo de busca em profundidade no grafo, guardando a lista de vértices percorridos a cada recorrência. A cada execução do algoritmo, duas ocorrências distintas são procuradas: a visita de um nó já visitado ou o encontro ao vértice destino. Na primeira situação um ciclo foi encontrado. A recorrência é interrompida para a sequência atual e a lista de vértices percorridos é submetida a um novo processamento, onde o último nó encontrado será procurado a partir do início da lista. Deseja-se guardar apenas os vértices que participam do ciclo, então quando o vértice

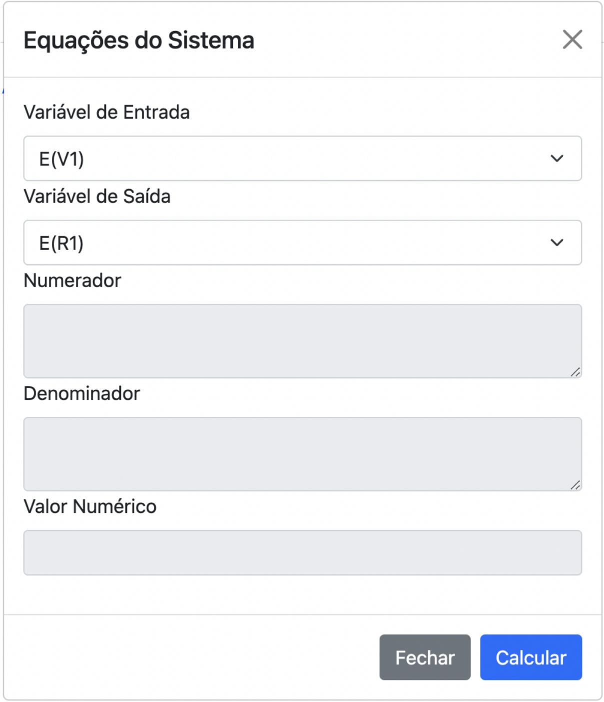


Figura 5.13: Janela de apresentação de resultados finais

que foi visitado duas vezes for encontrado, os elementos a partir dele são extraídos para uma nova lista, e essa é armazenada ao *array* de ciclos de primeira ordem. No segundo cenário, ao encontrar o nó destino, significa que um caminho direto foi encontrado, visto que após a obtenção de um ciclo o algoritmo se encerra para aquele caso. A lista de vértices é adicionada ao *array* de caminhos diretos e o processamento do algoritmo de busca continua sua operação, contando que o vértice destino também pode fazer parte de um ciclo.

Para adicionar um caminho direto ou um ciclo aos *arrays* do contexto, é necessário calcular o ganho total das arestas percorridas em cada cenário. Dessa forma, cada ocorrência de caminhos diretos ou ciclos devem estar armazenadas em estruturas de dados conforme mostra a figura 5.14.

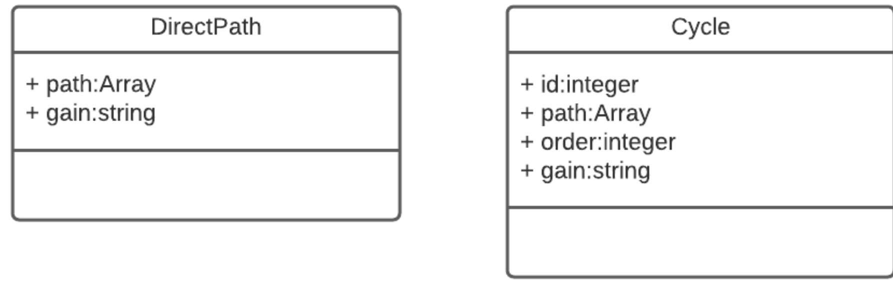


Figura 5.14: Diagrama de classes dos caminhos do DFS

O id dos ciclos é numérico e sequencial, gerado à partir da quantidade de elementos

na lista de ciclos do contexto acrescido de 1. Serve para execução do processo de calcular os laços de ordem superior que será explicado logo adiante. O ganho de cada elemento é calculado inicialmente apenas concatenando as variáveis de ganho de cada conexão do DFS como *strings* à sinais de multiplicação (*) entre eles.

Após encontrar os caminhos diretos e os laços de primeira ordem, também é preciso adicionar, caso existam, os ordem superior. Para isso, verifica-se se existe séries cíclicas que não possuam vértices em comum. A partir delas é gerado um novo elemento com ordem incrementada a 1, o ganho do novo objeto sendo a concatenação da variável ganho dos dois elementos envolvivos com o sinal de multiplicação e id também formado pela junção dos ids dos dois objetos *Cycle* com o simbolo -"entre eles. Para evitar que dois novos elementos de ordem superior, com mesmos ciclos envolvidos sejam gerados, é realizada uma ordenação entre os ids. Dessa forma, ao analisar o ciclo 2 com o 3 ou o 3 com o 2, será gerado um id 2-3, se o 2-3 for comparado com o ciclo 1 para gerar um elemento de terceira ordem, o id formado será 1-2-3. e assim por diante. A geração de elementos acontecerá enquanto houver componentes de ordem n-1 a serem analisados, onde n é a ordem atual da iteração, sendo incrementada ao ser encerrado o processamento de todos os objetos de uma mesma ordem. Em outras palavras, o algoritmo se encerra quando não forem mais gerados nenhum novo elemento de ordem n.

A partir disso foi possível calcular o valor de Δ, percorrendo a lista de ciclos agrupados por ordem. Os ganhos de cada um foram sendo concatenados com os sinais das operações de soma e subtração, de modo que à cada grau de ordem dos ciclos, a operação seria alternada, de acordo com a equação 4.20.

Para calcular o somatório dos caminhos diretos, foi preciso encontrar, para cada um deles o valor de Δ_k. Para isso foi inicializada uma variável *deltaKResult* com a string '1', e percorrida a lista de ciclos de primeira ordem para procurar ciclos que não possuíam vértices em comum com o caminho direto. Ao encontrar, o seu ganho era concatenado com a variável *deltaKResult* juntamente com o sinal de subtração para decrementar seu valor de 1. Assim foi possível finalizar o somatório dos caminhos diretos. Basta percorrer a lista, no contexto, e para cada caminho concatenar o seu valor de ganho multiplicado com Δ_k à uma variável resultante, inicialmente vazia.

Restava apenas realizar a operação polinomial para apresentar os resultados ao usuário. Para esta tarefa, foi preciso utilizar bibliotecas Javascript, pois o processo de simplificar expressões polinomiais pode se tornar bem complexo, fugindo do propósito deste trabalho. Dessa forma em um primeiro momento foi utilizada uma biblioteca chamada "Nerdamer", que apresenta ótimos resultados em termos de simplificação e permite separar o numerador e o denominador da função de transferência resultante sem muitas dificuldades. No entanto, foi percebido uma falha ao simplificar expressões quando Δ assumia valor unitário, inviabilizando o uso da solução nestes casos.

Uma segunda biblioteca chamada "Math.js"foi encontrada como solução para os

cenários não resolvidos pela "Nerdamer", porém o seu nível de simplificação não era tão efetivo, deixando a desejar no tratamento de funções polinomiais mais extensas. Pensou-se então em juntar as duas alternativas de modo que a Math.js atuasse apenas no cenário problemático para o Nerdamer, que seria o preferido para realizar simplificações e separar numerador e denominador do resultado polinomial final. Também foi utilizada para o cálculo da função de transferência numérica, na qual não apresentou nenhum tipo de problema ao realizar a operação de substituição das incógnitas pelos valores aplicados aos componentes na etapa de desenho do sistema físico.

Após resolver este último problema, os campos inferiores da tela mostrada na figura 5.13 foram preenchidos com os resultados finais obtidos pela regra de Mason.

Capítulo 6

Resultados

Neste capítulo serão apresentados o resultados obtidos com a construção do AutoTF. Em um primeiro momento, serão mostradas as funcionalidades presentes no ambiente desenvolvido. O sistema será submetido à alguns desenhos de circuitos com configurações confusas de serem organizadas na tela. Assim poderemos analisar os resultados oriundos das estratégias adotadas no capítulo 5, levando as técnicas de controle de colisão ao limite. A capacidade de escolha do tipo de ligação à depender da posição dos objetos também deve ser observada, visto que isso contribui para melhoria da interação com o usuário, mostrando que, apesar de não ser o objetivo principal do projeto, é de grande importância para garantir uma boa experiência ao modelar sistemas na plataforma.

Em um segundo momento poderemos visualizar o ambiente de apresentação dos resultados pelo AutoTF, observando a fundo a clareza dos dados que são mostrados na tela. Esse é um passo importante para mostrar que o projeto, de fato, consegue exibir corretamente para o usuário os procedimentos adotados para construção da função de transferência.

Por fim, colocaremos em evidência alguns resultados finais apresentados pelo AutoTF em comparação com os resultados dos mesmos modelos obtidos de forma manual, baseado em exemplos da literatura. A partir dessa abordagem, será possível testar se o AutoTF apresenta resultados confiáveis, garantindo que protótipos mais complexos de difícil verificação podem ser modelados com mesma precisão.

6.1 Organização do diagrama do protótipo

Nesta etapa do trabalho, devemos considerar que os desenhos apresentados à seguir dificilmente serão um cenário real, visto que o projetista normalmente elaboraria o seu projeto com um conceito mínimo de organização. No entanto, devemos pensar nos casos mais críticos para exemplificar as funcionalidades do sistema.

Serão utilizados quatro modelos diferentes. Os dois primeiros serão para apresentar o funcionamento do controle de colisão entre uma conexão e um componente, e os outros

dois para mostrar o cruzamento de linhas de conexões diferentes, sendo necessário o aparecimento do *bypass* para indicar que aquele ponto de intersecção não se trata de uma junta.

6.1.1 Cenário 1

O primeiro cenário consiste em uma sistema com três resistores em série ligados à uma fonte. O resistor R3 será aproximado dos outros dois, de modo que em uma uma situação onde o controle de colisão não atue, o resistor R3 fique por cima da linha que conecta o resistor R1 com o R2, confundindo o usuário, visto que a imagem passará a impressão de que o resistor R3 possa estar ligado diretamente com R1.

Analisando a figura 6.1, é possível ver que o que acontece uma mudança no tipo de conexão entre R1 e R2, representado por R4 e R5 na figura à direita. A conexão que antes possuía apenas 1 reta (uma única linha) passa para uma ligação de 3 retas para evitar colidir com o resistor R6 (representando o R3) quando ele passa a ocupar uma posição próxima a um limite mínimo estabelecido para margem das linhas de conexão.

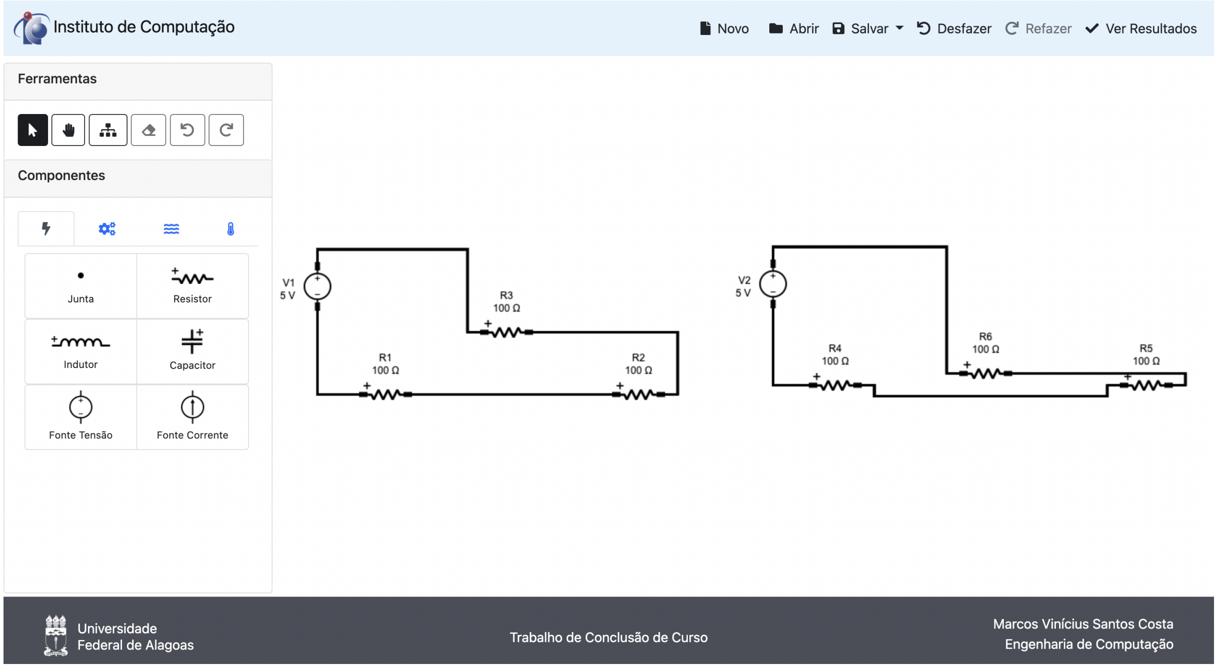


Figura 6.1: Sistema 1 sem colisões detectadas à esquerda. Controle de colisão atuando sobre o sistema, à direita

6.1.2 Cenário 2

O segundo cenário de simulação foi projetado para ser observada uma resposta do controle de colisão sem alterar o tipo de ligação entre os componentes. Foram colocados dois resistores (R1 e R4) em série com uma associação de resistores em paralelo (R2 e R3).

O objetivo é posicionar o resistor R3 em uma posição inferior até que seja observada a atuação do controle de colisão.

Como é possível observar na figura 6.2, o sistema detecta a eminente colisão e afasta a ligação entre o último resistor e a fonte, representados à direita por R8 e V2, respectivamente, de modo que o componente não consiga alcançar a linha da conexão. Observamos então que o controle de colisão atua diretamente na linha anterior, através de um algoritmo de busca que procura o fator mais próximo para incrementar (ou decrementar) do seu tamanho, fazendo com que a em que a linha que colidiria com R7 mude sua posição.

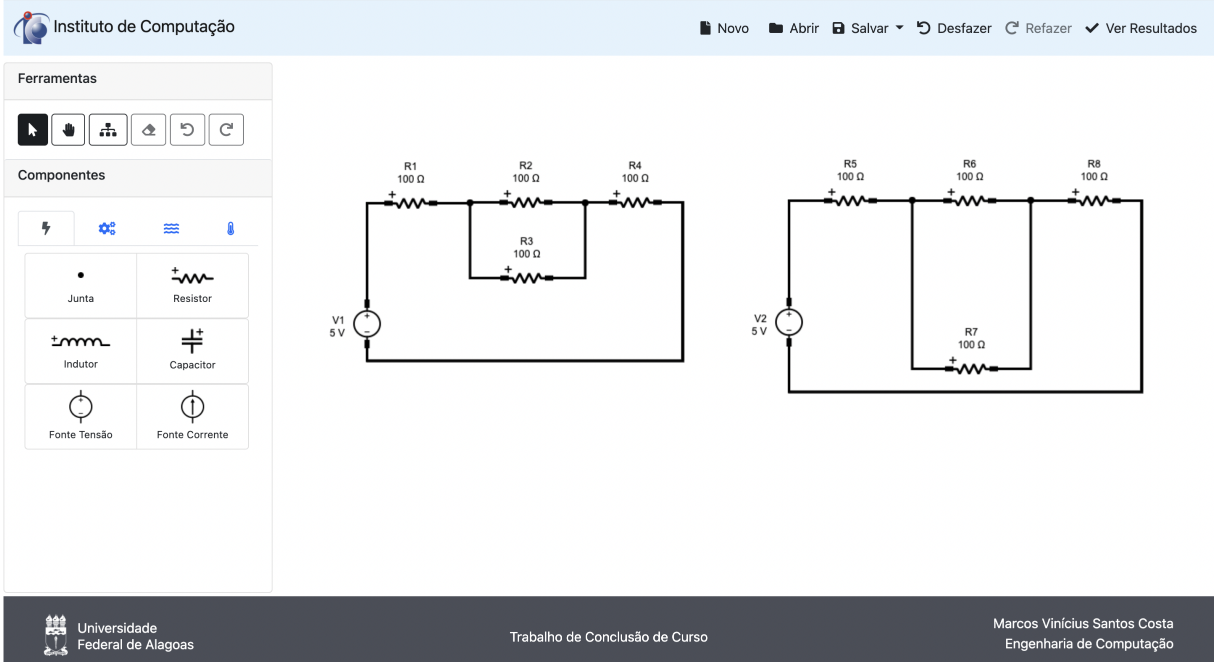


Figura 6.2: Sistema 2 sem colisões detectadas à esquerda. Controle de colisão atuando sobre o sistema, à direita

6.1.3 Cenário 3

Este caso é uma continuação do cenário anterior, com as mesmas configurações representadas pela fonte V1 ligada aos resistores R1 e R4, que estão em série com R2 e R3 em paralelo. O objetivo é continuar levando o resistor R3 para baixo até um certo ponto em que a distância mínima da linha com a sua posição adequada deixará de ser abaixo do componente que está sendo movimentado.

Ao observar a figura 6.3 percebemos que o comportamento do controle de colisão ocorre exatamente como esperado. A posição mais adequada para a linha colidente passa a ser acima do resistor. E nesse momento conseguimos observar a atuação do sistema de colisão entre conexões. Dois ícones de *bypass* são necessárias no novo sistema para para mostrar que R3 não está diretamente ligado à fonte V1. Como as ligações dos resistores

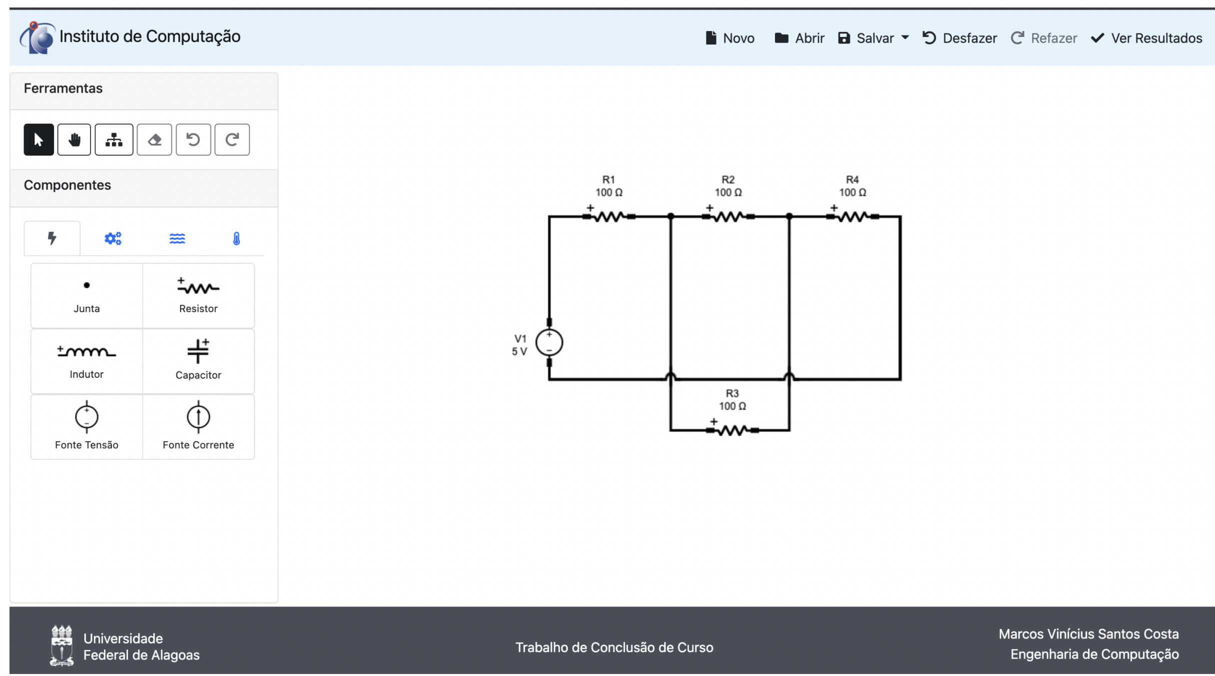


Figura 6.3: Sistema 3 mostrando funcionamento dos controles de colisão entre conexões

em paralelo foram feitas por último nesse modelo, elas ficaram por baixo a linha que liga R3 à V1.

6.1.4 Cenário 4

Neste último cenário, foi montado um circuito com componentes em posições mais críticas, com objetivo de avaliar a escolha para cada tipo de conexão pelo AutoTF, bem como verificar se é possível fazer uma boa interpretação do protótipo mesmo com muitas conexões entrelaçadas. A configuração consiste em uma fonte V1, e três resistores (R1, R2 e R3) ligados em série, sendo este último rotacionado 90º para a direita.

Analisando a figura 6.4, podemos verificar a atuação do controle de colisão entre conexões em três localizações diferentes. A presença dos *bypasses* possibilita que seja identificada a correta ligação de cada componente, mesmo em um cenário mais crítico. Além disso, o controle de colisão em objetos também atua na conexão entre R3 e V1, evitando a colisão com R1. Eventualmente, caso existam muitos componentes em posições desordenadas, com muitas juntas envolvidas e componentes rotacionados de forma desnecessária, ficaria muito complicado compreender o que está acontecendo. Mas como mencionado anteriormente, uma situação desse tipo se torna improvável e descartada desde que o usuário tenha um padrão de organização estabelecido, assim como ao utilizar qualquer outro *software* de simulação.

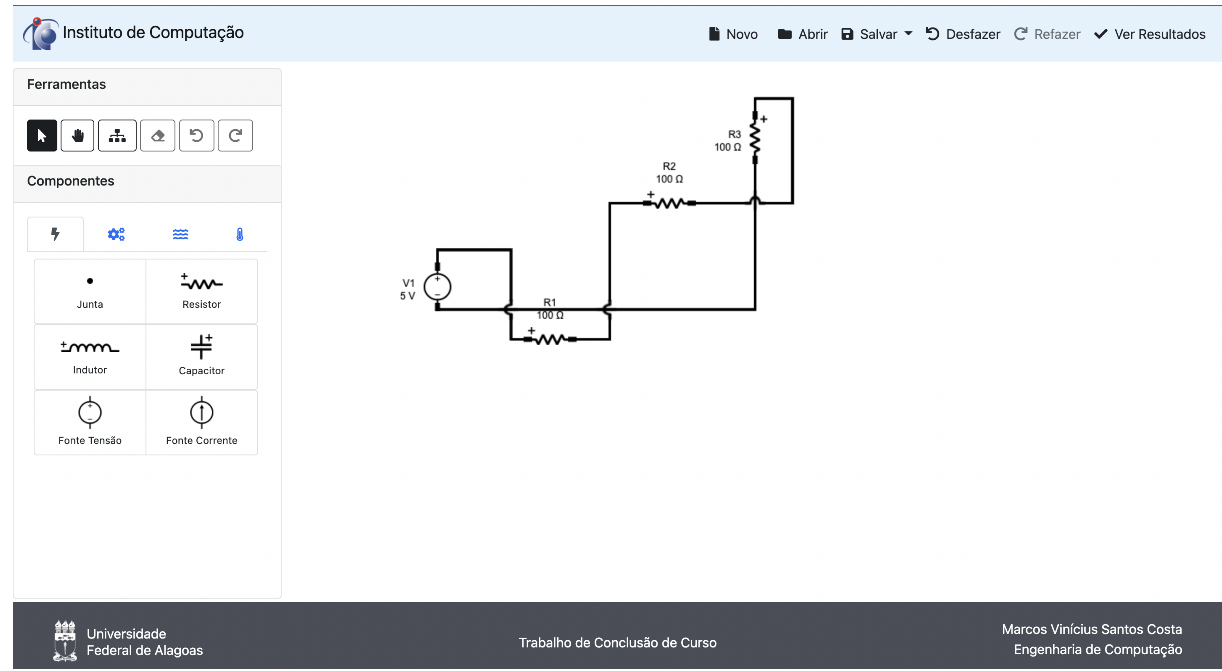


Figura 6.4: Sistema 4 mostrando funcionamento dos controles de colisão evitando que as ligações trombe com outras conexões ou objetos

6.2 Tela de Resultados

Um dos objetivos propostos para este trabalho é fazer com que o sistema AutoTF apresente resultados de forma organizada e coesa, deixando o usuário a par de todas as etapas para construção da função de transferência final. Para realizar essa tarefa, foi pensado em como a interface Web se adapta à esses tipos de exibição, sempre com foco no usuário. Como relatado no capítulo 5, o Canvas mais uma vez de mostrou como um recurso útil para que fosse possível construir o grafo de ligação e o diagrama de fluxos.

A figura 6.5 mostra que a interface do grafo de ligação e do DFS são baseadas na mesma metodologia de construção do desenho do circuito, mas de forma mais simples, uma vez que não é mais necessário utilizar os diferentes tipos de conexões dependentes da posição dos dois objetos que estão sendo conectados. Mesmo assim, no grafo de ligação e na DFS existe uma particularidade nas conexões: elas devem ser direcionadas. Por esse motivo os pesos dos arcos foram implementados como novos componentes, quadriculados com uma letra no centro indicando a sua natureza. O layout foi projetado para que o usuário possa mover cada um dos elementos do grafo, ou toda a área de trabalho, apenas com o uso do mouse. Assim ele pode organizar o grafo da forma que achar melhor para facilitar sua interpretação.

É importante ressaltar que uma lógica simples é utilizada para tentar organizar as posições iniciais dos elementos dos grafos. Mas ainda sim é algo muito primitivo e precisa que seja implementado um algoritmo de organização de um grafo plano para resolver definitivamente este problema.

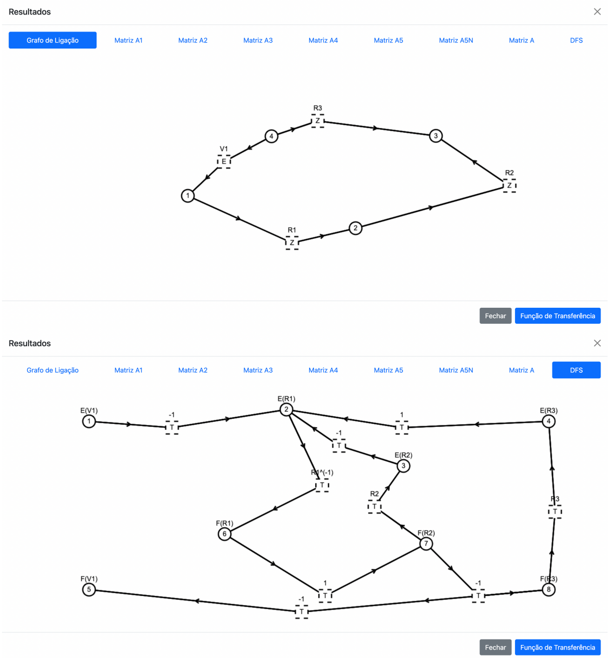
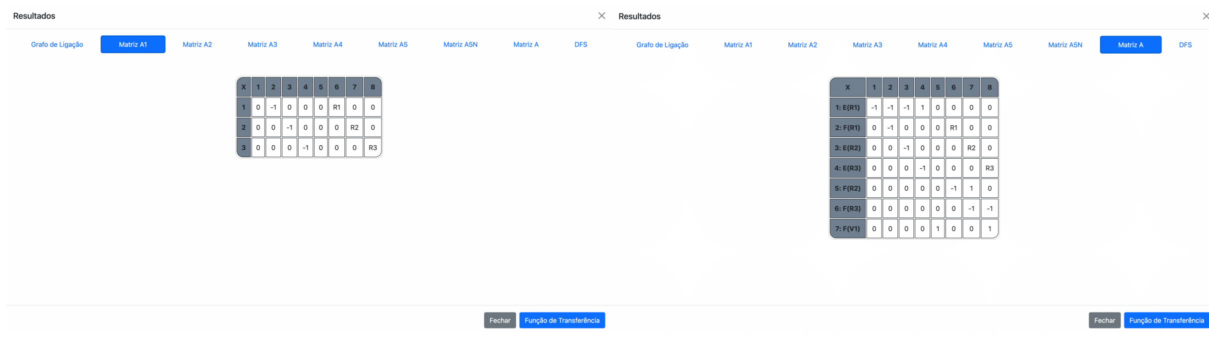


Figura 6.5: Janela de apresentação do grafo de ligação do AutoTF acima, DFS abaixo

O mesmo nível de apresentação foi almejado para exibição das tabelas de conectividade. Mesmo já tendo as informações em estruturas de dados internas no AutoTF, foi optado por exibir ao usuário cada uma das matrizes necessárias para obtenção do diagrama de fluxo de sinais, conforme mostrado na figura 6.6. No entanto, para a matriz A resultante, foi adicionado a cada uma de suas linhas a variável encontrada no algoritmo de busca. Assim, se o projetista desejar, ele mesmo pode realizar os cálculos à partir da matriz para obter a DFS.



X	1	2	3	4	5	6	7	8
1	0	-1	0	0	0	R1	0	0
2	0	0	-1	0	0	0	R2	0
3	0	0	0	-1	0	0	0	R3

X	1	2	3	4	5	6	7	8
1: E(R1)	-1	-1	-1	1	0	0	0	0
2: F(R1)	0	-1	0	0	0	R1	0	0
3: E(R2)	0	0	-1	0	0	0	R2	0
4: E(R3)	0	0	0	-1	0	0	0	R3
5: F(R2)	0	0	0	0	0	-1	1	0
6: F(R3)	0	0	0	0	0	0	-1	-1
7: F(V1)	0	0	0	0	1	0	0	1

Figura 6.6: Janela de apresentação da matriz A1 à esquerda, matriz A à direita

6.3 Análise da Função de Transferência

O objetivo final da solução proposta por esse trabalho é apresentar funções no domínio de Laplace que representasse o comportamento do sistema a partir de um elemento de entrada e um de saída. A integridade e veracidade das funções resultantes seriam então essenciais

para comprovar que este trabalho obteve um bom resultado e o sistema desenvolvido pode ser utilizado, de fato, para cumprir este propósito. Dessa forma, foram elaborados alguns cenários de circuitos elétricos diferentes, compostos com fontes de tensão/corrente, resistores, capacitores e indutores, para avaliarmos o procedimento lógico e matemático adotado pelo AutoTF utilizado para obtenção dessas equações.

Esta análise será composta por três casos distintos, onde nos dois primeiros utilizaremos modelos mais conhecidos na literatura, e no último um sistema mais complexo será modelado para assim podermos comparar, em cada caso, os resultados oriundos de mesmas portas de entrada e saída.

6.3.1 Cenário 1

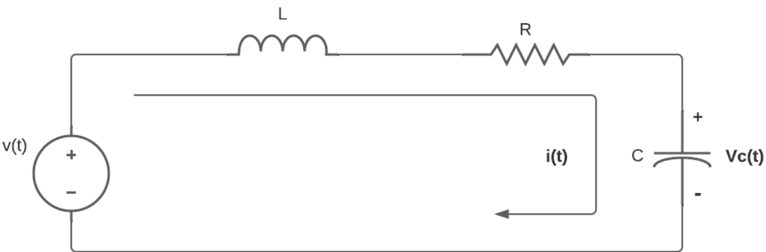


Figura 6.7: Circuito RLC em série

A figura 6.7 mostra a configuração inicial escolhida. O circuito RLC em série é uma boa alternativa para validar se as características dos componentes básicos de circuitos elétricos estão sendo respeitadas. Trata-se de um sistema já conhecido, se mostrando adequado para iniciar as verificações em torno dos resultados finais apresentados pela solução deste trabalho. O circuito é formado por apenas uma única malha, portanto, o cálculo para obtenção da função de transferência entre uma porta de entrada e uma porta de saída do sistema é simples. A função de transferência entre a variável de esforço da fonte (tensão de entrada) e a tensão no capacitor é obtida de acordo com o procedimento abaixo.

A lei de Kirchhoff das tensões nos diz que a soma de todas as tensões existentes em uma malha deve ser igual a 0. Dessa forma, igualamos a tensão da fonte com a tensão dos três outros componentes:

$$V(s) = LsI(s) + RI(s) + \frac{1}{Cs}I(s) \tag{6.1}$$

Para a tensão no capacitor, temos que:

$$V_c(s) = \frac{1}{Cs}I(s) \tag{6.2}$$

Para chegar à função de transferência desejada, precisamos encontrar a relação entre $V_c(s)$ e $V(s)$. Logo:

$$\begin{aligned}
G(s) = \frac{V_c(s)}{V(s)} &= \frac{\frac{1}{Cs}I(s)}{LsI(s) + RI(s) + \frac{1}{Cs}I(s)} \\
&= \frac{\frac{1}{Cs}I(s)}{I(s)(Ls + R + \frac{1}{Cs})} \\
&= \frac{\frac{1}{Cs}I(s)}{I(s)(Ls + R + \frac{1}{Cs})} \\
&= \frac{\frac{1}{Cs}}{\frac{LCs^2+RCs+1}{Cs}} \\
G(s) &= \frac{1}{LCs^2 + RCs + 1}
\end{aligned} \tag{6.3}$$

Comparando com os resultados apresentados pelo AutoTF na figura 6.8, podemos ver que o valor do numerador e do denominador são equivalentes ao resultado da equação 6.3.

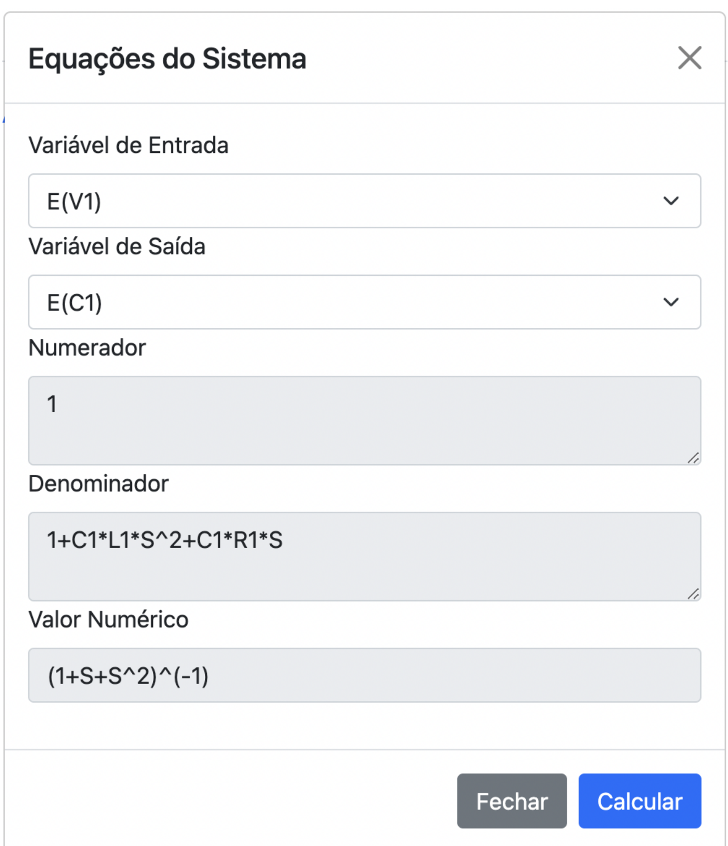


Figura 6.8: AutoTF - Resultado para circuito RLC em série

6.3.2 Cenário 2

No segundo cenário, o objetivo foi testar o efeito do paralelismo em um sistema elétrico. Além disso, era necessário avaliar também como a solução se comportaria em cenários com mais de uma malha. Dessa forma, o circuito da figura 6.9 foi o escolhido para ser comparado. Da mesma forma que o anterior, foram utilizados todos os tipos de elementos armazenadores e dissipadores disponíveis, de modo que o indutor ficasse em paralelo com o capacitor e um dos resistores.

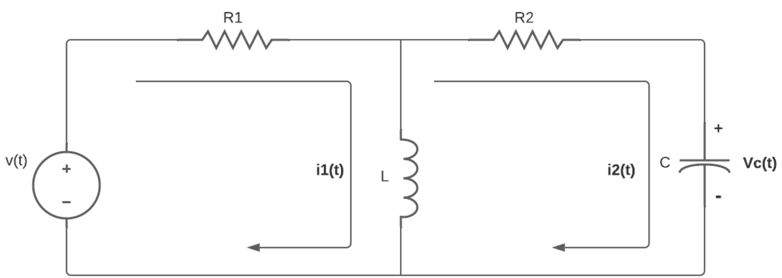


Figura 6.9: Circuito elétrico com duas malhas

A função de transferência desejada deve ser calculada à partir da tensão da fonte como variável de entrada, e a corrente no indutor como variável de saída. Os cálculos necessários para este exemplo também não foram complicados, mas por ter uma malha a mais que o modelo anterior, se mostraram um pouco mais extensos.

Partindo mais uma vez da lei das malhas de Kirchhoff, temos que:

$$R_1 I_1(s) + LsI_1(s) - LsI_2(s) = V(s) \tag{6.4}$$

$$LsI_2(s) + R_2 I_2(s) + \frac{1}{Cs} I_2(s) - LsI_1(s) = 0 \tag{6.5}$$

Para encontrar $I_1(s)$, basta isolarmos essa variável na equação 6.5 como segue:

$$I_1(s) = \frac{I_2(s)(Ls + R_2 + \frac{1}{Cs})}{Ls} \tag{6.6}$$

Substituindo 6.6 em 6.4:

$$V(s) = I_2(s) \left(\frac{Ls + R_2 + \frac{1}{Cs}}{Ls} (R_1 + Ls) - Ls \right) \tag{6.7}$$

Agora, basta reorganizarmos os termos de 6.7 para obter a função de transferência desejada:

$$
\begin{aligned}
\frac{V(s)}{I_2(s)} &= \frac{LsR_1 + L^2s^2 + R_1R_2 + LsR_2 + R_1\frac{1}{Cs} + Ls\frac{1}{Cs} - L^2s^2}{Ls} \\
&= \frac{R_1Ls + R_1R_2 + LsR_2 + \frac{R_1}{Cs} + \frac{L}{C}}{Ls} \\
&= \frac{\frac{R_1LCs^2 + R_1R_2Cs + CR_2Ls^2 + R_1 + Ls}{Cs}}{Ls} \\
\frac{I_2(s)}{V(s)} &= \frac{CLs^2}{LR_1Cs^2 + R_1R_2Cs + CR_2Ls^2 + R_1 + Ls}
\end{aligned}
\tag{6.8}
$$

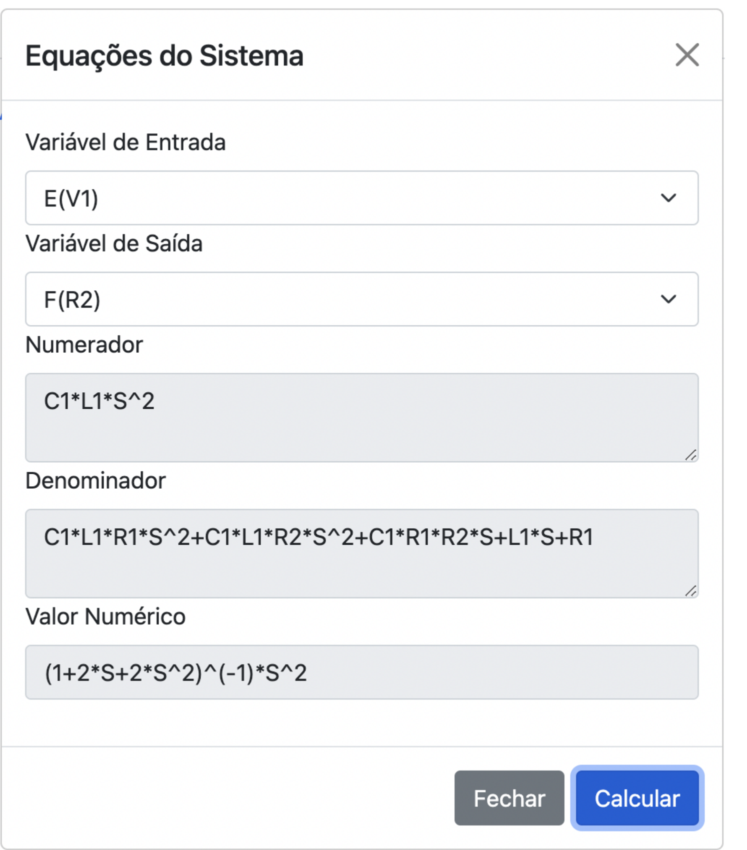


Figura 6.10: AutoTF - Resultado para circuito elétrico com duas malhas

6.3.3 Cenário 3

Para o terceiro cenário, o AutoTF foi submetido à um sistema um pouco mais complexo, com 3 malhas e uma fonte de corrente para que seja verificado se seu comportamento pôde ser modelado corretamente. A fonte de corrente possui uma propriedade diferente da fonte de tensão na geração do grafo de ligação, cujas variáveis de esforço possuem sentidos invertidos. Além disso, a variável de entrada deixa de ser uma variável de esforço e passa a ser uma variável de fluxo. A solução deve ser capaz de identificar todas essas características. O circuito elétrico mostrado na figura 6.11 foi o escolhido para esta veri-

ficação de modo que fosse obtida a função de transferência à partir da corrente da fonte como variável de entrada e a tensão no capacitor C como variável de saída.

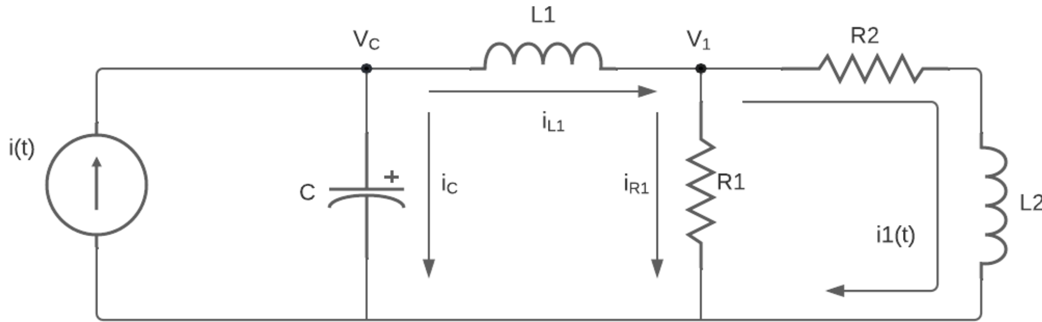


Figura 6.11: Circuito elétrico com fonte de corrente e 3 malhas

Para iniciar a análise do circuito proposto, utilizaremos a Lei de Kirchhoff dos nós para obter as equações do sistema. À partir dos nós V_C e V_1, igualando a soma das correntes de entrada nos nós com a soma das correntes de saída, temos as equações 6.9 e 6.10.

$$I(s) = I_{L1}(s) + I_C(s) \tag{6.9}$$

$$I_{L1}(s) = I_{R1}(s) + I_1(s) \tag{6.10}$$

Sabemos também que as correntes podem ser encontradas através do quociente entre a diferença de potencial entre os nós, pela impedância dos elementos pelos quais ela passa. Dessa forma chegamos à:

$$I_C(s) = \frac{V_C(s)}{C^{-1}s^{-1}} = CsV_C(s) \tag{6.11}$$

$$I_{L1}(s) = \frac{V_C(s) - V_1(s)}{L1s} \tag{6.12}$$

$$I_{R1}(s) = \frac{V_1(s)}{R1} \tag{6.13}$$

$$I_1(s) = \frac{V_1(s)}{R2 + L2s} \tag{6.14}$$

Substituindo as equações 6.12, 6.13 e 6.14 em 6.10 para encontrar o valor de $V_1(s)$, temos que:

$$\frac{V_C(s) - V_1(s)}{L1s} = \frac{V_1(s)}{R1} + \frac{V_1(s)}{R2 + L2s}$$

$$V_C(s) - V_1(s) = L1s\left(\frac{(R2 + L2s)V_1(s) + R1V_1(s)}{R1(R2 + L2s)}\right)$$

$$R1(R2 + L2s)V_C(s) - R1(R2 + L2s)V_1(s) = L1s(R2 + L2s)V_1(s) + L1sR1V_1(s)$$

$$R1(R2 + L2s)V_C(s) = V_1(s)(L1s(R2 + L2s) + L1sR1 + R1(R2 + L2s))$$

$$V_1(s) = V_C(s)\left(\frac{R1(R2 + L2s)}{L1s(R1 + R2 + L2s) + R1(R2 + L2s)}\right) \tag{6.15}$$

Substituindo o valor de $V_1(s)$ na equação 6.12, podemos descobrir o valor de $I_{L1}(s)$:

$$\begin{aligned} I_{L1}(s) &= \frac{V_C(s) - V_C(s)\left(\frac{R1(R2+L2s)}{L1s(R1+R2+L2s)+R1(R2+L2s)}\right)}{L1s} \\ &= \frac{V_C(s)}{L1s}\left(1 - \frac{R1(R2 + L2s)}{L1s(R1 + R2 + L2s) + R1(R2 + L2s)}\right) \\ &= \frac{V_C(s)}{L1s}\left(\frac{L1s(R1 + R2 + L2s)}{L1s(R1 + R2 + L2s) + R1(R2 + L2s)}\right) \\ &= V_C(s)\left(\frac{R1 + R2 + L2s}{L1s(R1 + R2 + L2s) + R1(R2 + L2s)}\right) \end{aligned} \tag{6.16}$$

Agora, substituindo as equações 6.11 e 6.16 em 6.9:

$$\begin{aligned} I(s) &= V_C(s)\left(\frac{R1 + R2 + L2s}{L1s(R1 + R2 + L2s) + R1(R2 + L2s)}\right) + CsV_C(s) \\ &= V_C(s)\left(\frac{R1 + R2 + L2s}{L1s(R1 + R2 + L2s) + R1(R2 + L2s)} + Cs\right) \\ &= V_C(s)\left(\frac{L1Cs^2(R1 + R2 + L2s) + R1Cs(R2 + L2s) + R1 + R2 + L2s}{L1s(R1 + R2 + L2s) + R1(R2 + L2s)}\right) \end{aligned} \tag{6.17}$$

Para encontrar a função de transferência desejada, precisamos encontrar o valor de $\frac{V_C(s)}{I(s)}$. Logo:

$$\begin{aligned} \frac{V_C(s)}{I(s)} &= \frac{L1s(R1 + R2 + L2s) + R1(R2 + L2s)}{L1Cs^2(R1 + R2 + L2s) + R1Cs(R2 + L2s) + L2s + R1 + R2} \\ &= \frac{L1L2s^2 + L1R1s + L1R2s + L2R1s + R1R2}{CL1L2s^3 + CL1R1s^2 + CL1R2s^2 + CL2R1s^2 + CR1R2s + L2s + R1 + R2} \end{aligned} \tag{6.18}$$

Ao observar a expressão final na equação 6.18 que representa à função de transferência $\frac{V_C(s)}{I(s)}$, vemos que o resultado é o mesmo apresentado pelo AutoTF, como mostra a figura 6.12.

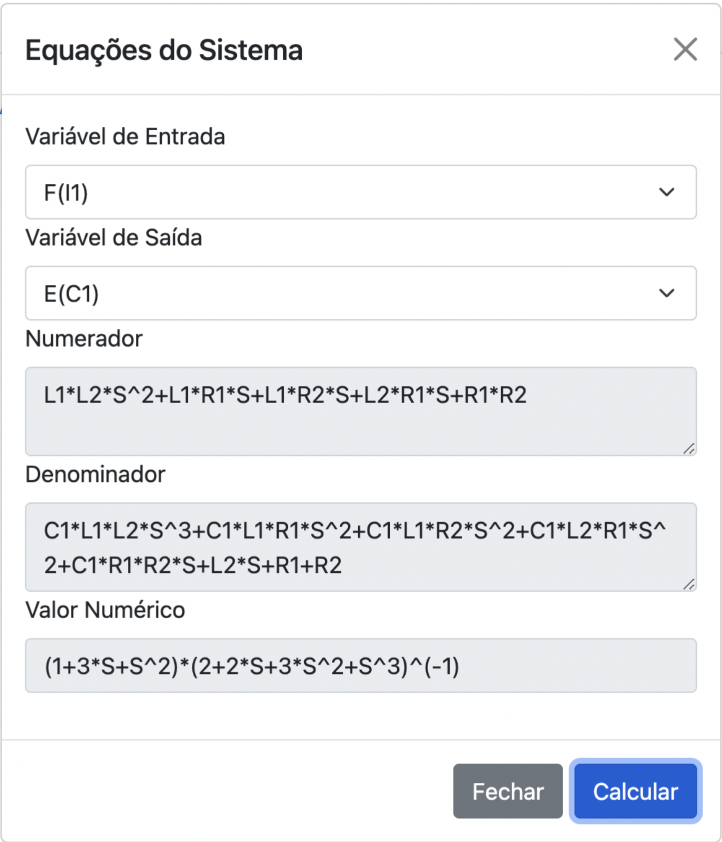


Figura 6.12: AutoTF - Resultado para circuito elétrico com fonte de corrente e 3 malhas

6.4 Sistemas sem solução

No processo de construção do diagrama de fluxo de sinais foi mostrado que o AutoTF utiliza a matriz A, de interconexão entre os componentes para que sejam estabelecidas as tramitâncias do diagrama, de modo que cada linha de A seja correspondente a uma das variáveis de saída do sistema. No entanto, ao colocar duas fontes de tensão em paralelo ou duas fontes de corrente em série, os resultados apresentados pela matriz resultante impedem que o algoritmo de busca consiga associar as variáveis de saída às linhas de A. Isso pode ser explicado pelo teorema da superposição, que diz que a tensão ou corrente em cada componente é igual a soma das tensões ou correntes produzidos independentemente em cada fonte. A figura 6.13 ilustra o cenário para cada um dos tipos de fontes elétricas.

No primeiro caso, aplicando o teorema da superposição, seria correto reescrever a primeira fonte de esforço com o valor da soma algébrica das duas fontes, fechando um curto-circuito na segunda. Dessa forma, existiriam dois valores de esforço possíveis para o

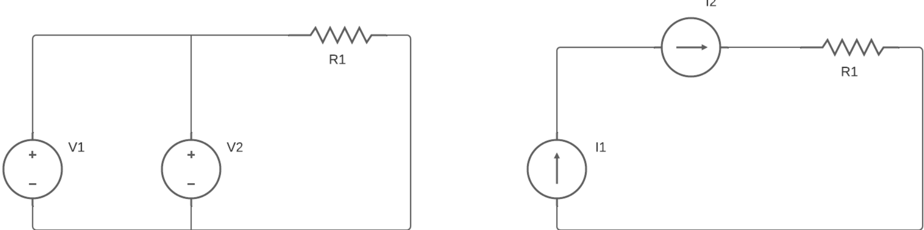


Figura 6.13: Exemplos de circuitos sem solução no AutoTF

resistor R1: V1+V2 e 0. Esse cenário invalida as equações lineares fornecidas pela matriz A, de modo que uma mesma variável possuiria duas funções diferentes para representá-la. O comportamento é análogo às fontes de fluxo, que ao somar as variáveis de fluxo e atribuir na primeira, abrindo o circuito na segunda fonte, o resistor R1 possuiria dois possíveis valores para sua variável de fluxo, bem como duas equações que o representasse na matriz. Isso ocasionaria uma falha no algoritmo de busca de variáveis, impedindo que cada linha fosse associada a uma variável de saída. O DFS então seria gerado apenas com os vértices, tornando-se possível sua aplicação na regra de Mason. A figura 6.14 apresenta a tela de resultados do AutoTF com esse cenário, indicando para o usuário que a função de transferência não poderá ser gerada.

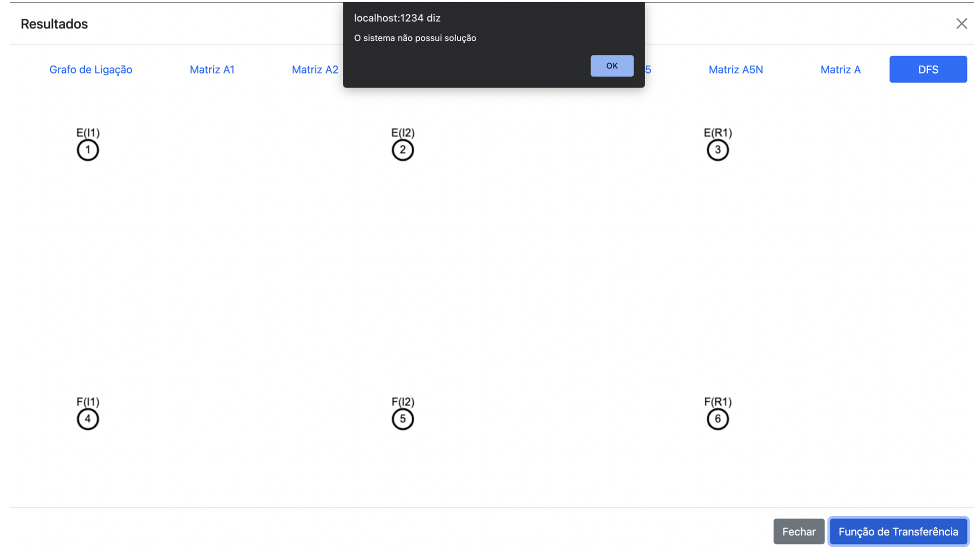


Figura 6.14: Tela de resultados no AutoTF - Cenário sem solução

6.5 Discussão

Algumas características puderam ser percebidas ao analisar os resultados de cada seção. O AutoTF se mostra como um projeto capaz de atrair o usuário interessado em projetar sistemas físicos a partir da facilidade em utilizá-lo. Interfaces modernas, de fácil

reconhecimento permitem que os projetistas sejam guiados ao procedimento padrão executado pelo simulador, dando ferramentas para edição de projeto em caso de não atingir o objetivo desejado.

Embora não sejam essenciais para o cálculo da função de transferência, as ferramentas implementadas na etapa de desenho do sistema se mostraram bastante úteis para manter a organização do projeto, evitando possíveis falhas de planejamento no processo de modelagem. Além disso, como é um recurso voltado para desenhos de diagramas e modelos, pode ser reutilizado em outros projetos através da implementação de uma biblioteca com a lógica aplicada neste trabalho, apesar de ainda existir muito espaço para possíveis melhorias futuras, automatizando o processo de escolher qual o melhor tipo de ligação para cada uma das combinações de posicionamento dos elementos.

Quanto à geração da função de transferência, o AutoTF cumpriu bem as metas estabelecidas, onde à partir de exemplos, ainda que básicos considerando a dimensão de projetos de sistemas físicos reais, se mostra bastante confiável, apresentando resultados precisos, idênticos aos resultados calculados manualmente e com funções finais simplificadas e fáceis de serem verificadas. Disponibilizar a forma numérica da função de transferência se faz útil para visualização do comportamento do sistema através de gráficos.

Por fim, pode-se dizer que a solução apresentada neste trabalho representa uma ótima aplicação de conceitos de teoria dos grafos, teorias de modelagem de sistemas no contexto de controle, com a aplicação da Regra de Mason e programação orientada a objetos voltadas para o cenário Web, que utiliza uma linguagem com um bom poder de processamento, fazendo com que o processo de geração do modelo matemático ocorra dentro de uma janela de tempo razoável, mesmo para modelagens de grandes proporções.

Conclusão

Este trabalho apresentou o AutoTF, um ambiente Web de modelagem de sistemas físicos lineares como uma ferramenta capaz de auxiliar o projetista de controle na etapa de prototipagem de seus trabalhos. Conseguir visualizar todo o processo acontecendo em cada etapa, reflete na confiabilidade dos resultados, tornando até mesmo a correção de erros uma tarefa mais fácil durante a construção do código e modelagem do sistema. A necessidade de usar algoritmos já difundidos na literatura para resolver problemas específicos da construção do grafo de ligação e do diagrama de fluxo de sinais, ou até mesmo na aplicação da regra de Mason nos mostra a importância dessas contribuições para o cenário geral da computação, assim como sua efetividade.

A possibilidade de escrever um simulador de sistemas físicos em uma linguagem poderosa como o Javascript se mostrou bastante satisfatória por um lado, mas bem desafiadora por outro. Montar uma área para desenho de circuitos pensando em todas as possíveis interações que o usuário possa vir a ter com a página e elaborar estratégias para organização e disposição dos componentes à medida que vão sendo adicionados, mantendo o projetista a par do que está acontecendo, não é uma tarefa trivial. Prova disso é que aplicações capazes de fazer tarefas similares normalmente são pagas, e muitas delas não têm exito em tratar algumas situações básicas de colisão que podem vir a acontecer.

Tornar possível a disponibilidade deste projeto na Internet é de fundamental importância para reduzir a quantidade de obstáculos existentes na utilização de um recurso de mesma categoria. Hoje em dia é bem mais fácil acessar o navegador para realizar uma operação simples em uma imagem do que baixar um *software* dedicado para realizar a tarefa, caso ele seja compatível com o sistema operacional. Apesar de o AutoTF ainda não estar em formato responsivo, permitindo sua execução em telas de *smartphones*, ele foi implementado com recursos que possibilitam essa questão.

Fica evidente que o projeto apresentado neste trabalho se trata de um modelo inicial capaz de possuir muitos outros recursos em momentos futuros. A adição mais componentes na paleta, sejam eles elétricos ou de outras naturezas disponíveis nas abas da interface gráfica: (Mecânico, hidráulico e térmico), além da implementação de uma versão responsiva fica também como sugestão para trabalhos posteriores.

As bibliotecas utilizadas para simplificação de equações polinomiais representam também um ponto a ser melhorado no AutoTF. Nenhuma das duas conseguem, sem

o auxílio da outra, representar uma solução ótima, que nos forneça as funções de transferência no formato desejado. A implementação de uma biblioteca matemática voltada exclusivamente para esse propósito se mostra um ponto de grande relevância no desenvolvimento deste projeto. No entanto, trata-se de algo muito extenso, talvez até mesmo um trabalho à parte, uma vez que simplificar expressões polinomiais pode ser um tarefa bastante complicada.

Por fim, a comparação de resultados do AutoTF com modelos existentes na literatura, nos permitiu garantir um bom nível de confiabilidade em relação as funções de transferência apresentadas por esse projeto. Mesmo com implementações diferentes em muitas etapas, ocasionando a geração DFS distintas para um mesmo sistema; apresentar a mesma função de transferência nos mostra que o objetivo principal deste trabalho foi alcançado.

Bibliografia

[BALIEIRO, 2015] BALIEIRO, R. (2015). *ESTRUTURA DE DADOS*. "SESES".

[Broenink, 1999] Broenink, J. F. (1999). Introduction to physical systems modelling with bond graphs. *SiE whitebook on simulation methodologies*, 31(2).

[Brown, 2001] Brown, F. T. (2001). *Engineering system dynamics a unified graph-centered approach*. Marcel Dekker, Inc.

[Burd et al., 1999] Burd, L. et al. (1999). Desenvolvimento de software para atividades educacionais. *Campinas: UNICAMP*.

[Carvalho and Itália, 2005] Carvalho, M. A. G. and Itália, J. N. (2005). Teoria dos grafos—uma introducao. *Universidade Estadual de Campinas - UNICAMP*.

[Costa, 2011] Costa, P. P. d. (2011). Teoria dos grafos e suas aplicações. *Universidade Estadual Paulista (UNESP)*.

[Eis and Ferreira, 2012] Eis, D. and Ferreira, E. (2012). *HTML5 e CSS3 com farinha e pimenta*. Lulu. com.

[El-Hajj and Kabalan, 1995] El-Hajj, A. and Kabalan, K. (1995). A transfer function computational algorithm for linear control systems. *Control Systems, IEEE*, 15:114 – 118.

[Farinelli, 2007] Farinelli, F. (2007). Conceitos básicos de programação orientada a objetos. *Instituto Federal Sudeste de Minas Gerais*.

[Feofiloff et al., 2011] Feofiloff, P., Kohayakawa, Y., and Wakabayashi, Y. (2011). Uma introdução sucinta à teoria dos grafos. *Universidade de São Paulo*.

[Fernandes, 2003] Fernandes, J. H. C. (2003). Qual a prática do desenvolvimento de software? *Ciência e Cultura*, 55(2):29–33.

[Ferraiolo et al., 2000] Ferraiolo, J., Jun, F., and Jackson, D. (2000). *Scalable vector graphics (SVG) 1.0 specification*. iuniverse Bloomington.

[Flanagan, 2004] Flanagan, D. (2004). *JavaScript: o guia definitivo*. Bookman Editora.

[Franklin et al., 2009] Franklin, G. F., Powell, J. D., and Emami-Naeini, A. (2009). *Sistemas de Controle para Engenharia*. Pearson Prentice Hall.

[Fulton and Fulton, 2013] Fulton, S. and Fulton, J. (2013). *HTML5 canvas: native interactivity and animation for the web*. "O'Reilly Media, Inc.".

[Gonçalves, 2018] Gonçalves, M. V. (2018). Programação orientada a objetos. *Revista Ada Lovelace*, 2:106–110.

[Gudwin et al., 1998] Gudwin, R. R., Shin-Ting, W., and Ricarte, I. L. (1998). Estruturas de dados.

[Jackson, 1975] Jackson, M. A. (1975). *Principles of program design*. Academic Press, Inc.

[Loudon, 2018] Loudon, K. (2018). Desenvolvimento de grandes aplicações web. *Revista Telfract*, 1(1).

[Lucchesi, 1979] Lucchesi, C. L. (1979). *Introdução à teoria dos grafos*. IMPA.

[Maitelli and SILVA, 2004] Maitelli, A. and SILVA, G. (2004). Um algoritmo para construÇÃo de diagramas de fluxo de sinal de sistemas fÍsicos. *Universidade Federal do Rio Grande do Norte*.

[Maitelli and SILVA, 2005] Maitelli, A. and SILVA, G. (2005). Um ambiente computacional para modelagem simbólica de sistemas físicos lineares utilizando grafos. *BRAZILIAN JOURNAL OF COMPUTERS IN EDUCATION*.

[Maluf, 2020] Maluf, A. S. (2020). Modelos dinâmicos aplicados a sistemas de manufatura utilizando grafos de ligação: proposta de modificações.

[Mansfield, 2005] Mansfield, R. (2005). *CSS web design for dummies*. John Wiley & Sons.

[Miletto and de Castro Bertagnolli, 2014] Miletto, E. M. and de Castro Bertagnolli, S. (2014). *Desenvolvimento de Software II: Introdução ao Desenvolvimento Web com HTML, CSS, JavaScript e PHP-Eixo: Informação e Comunicação-Série Tekne*. Bookman Editora.

[Mota, 2019] Mota, G. O. (2019). *Teoria dos grafos*. 1ª edição. Santo André: CMCC–Universidade Federal do ABC.

[Mowery and Shacham, 2012] Mowery, K. and Shacham, H. (2012). Pixel perfect: Fingerprinting canvas in html5. *Proceedings of W2SP*, 2012.

[Negrão, 2012] Negrão, C. G. (2012). Desenvolvimento de algoritmo para modelagem e simulaÇÃo de sistemas por grafos de ligaÇÃo. *Universidade Estadual Paulista*, page 143.

[Nise, 2013] Nise, N. S. (2013). *Engenharia de Sistemas de Controle*. LTC Rio de Janeiro.

[Ogata, 2010] Ogata, K. (2010). *Engenharia de Controle Moderno*. LTC Rio de Janeiro.

[OTSUKA and ZANELATO, 2012] OTSUKA, G. S. and ZANELATO, A. P. A. (2012). Programação orientada a objetos com a linguagem microsoft c-sharp. *Intertem@ s Negócios ISSN 1983-4462*, 9(9).

[Pérez Ibarra et al., 2021] Pérez Ibarra, S. G., Quispe, J. R., Mullicundo, F. F., and Lamas, D. A. (2021). Herramientas y tecnologías para el desarrollo web desde el frontend al backend. In *XXIII Workshop de Investigadores en Ciencias de la Computación (WICC 2021, Chilecito, La Rioja).*

[Pressman and Maxim, 2016] Pressman, R. S. and Maxim, B. R. (2016). *Engenharia de software-8ed*. McGraw Hill Brasil.

[Prestes, 2016] Prestes, E. (2016). Introdução à teoria dos grafos. *Universidade Federal do Rio Grande do Sul, Instituto de Informática, Departamento de Informática Teórica, Tech. Rep.*

[Rezende, 2006] Rezende, D. A. (2006). *Engenharia de software e sistemas de informação*. Brasport.

[Ricarte, 2001] Ricarte, I. L. M. (2001). Programação orientada a objetos: uma abordagem com java. *http://www. dca. fee. unicamp. br/cursos/PooJava/Aulas/poojava. pdf*, 29(10):2014.

[Ricarte, 2008] Ricarte, I. L. M. (2008). Estruturas de dados.

[Silva, 2005] Silva, G. A. (2005). Um ambiente computacional para modelagem simbólica de sistemas físicos lineares. *Universidade Federal do Rio Grande do Norte*, page 198.

[Silva, 2010] Silva, M. S. (2010). *JavaScript-Guia do Programador: Guia completo das funcionalidades de linguagem JavaScript*. Novatec Editora.

[Soares de Melo et al., 2014] Soares de Melo, G. et al. (2014). Introdução à teoria dos grafos. *Universidade Federal da Paraíba.*

[Souza, 2016] Souza, P. M. R. (2016). *Um estudo sobre padrões e tecnologias para o desenvolvimento web-front-end*. PhD thesis, Universidade Federal do Rio de Janeiro.

[Spurlock, 2013] Spurlock, J. (2013). *Bootstrap: responsive web development*. "O'Reilly Media, Inc.".

[STEMMER, 2001] STEMMER, D.-I. M. R. (2001). Das 5331-sistemas distribuídos e redes de computadores para controle e automação industrial. *UFSC, Departamento de Automação e Sistemas, Florianópolis*.

[Stroustrup, 1988] Stroustrup, B. (1988). What is object-oriented programming? *IEEE software*, 5(3):10–20.

[Viana, 2016] Viana, L. A. d. C. (2016). Árvore geradora com dependências mínima. *UNIVERSIDADE FEDERAL DO CEARA*.

[Wellstead, 1979] Wellstead, P. E. (1979). *Introduction to physical system modelling*. Academic Press.

[Wheatman and Xu, 2018] Wheatman, B. and Xu, H. (2018). Packed compressed sparse row: A dynamic graph representation. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE.

[WPInfo, 2010] WPInfo (2010). Usando svg vs. canvas: um breve guia. https://br.atsit.in/archives/34179. [Online; acessado em 27-03-2022].

[Zemel, 2015] Zemel, T. (2015). *Web Design Responsivo: páginas adaptáveis para todos os dispositivos*. Editora Casa do Código.