Trabalho de Conclusão de Curso

VaucangraphFx: Estudo e desenvolvimento de uma ferramenta de apoio na modelagem de grafos e autômatos.

Alex Vieira Alves

Orientação: Prof. Dr. Gedson Faria

Bacharelado em Sistemas de Informação

Campus de Coxim
Universidade Federal de Mato Grosso do Sul
25 de Junho de 2019

VaucangraphFx: Estudo e desenvolvimento de uma ferramenta de apoio na modelagem de grafos e autômatos.

Alex Vieira Alves

Bacharelado em Sistemas de Informação

Monografia apresentada à Banca Examinadora como exigência parcial para obtenção do título de Bacharel em Sistemas de Informação pela Universidade Federal de Mato Grosso do Sul, sob a orientação do Prof. Doutor Gedson Faria.

Coxim-MS
2019.

VaucangraphFx: Estudo e desenvolvimento de uma ferramenta de apoio na modelagem de grafos e autômatos.

25 de Junho de 2019.

Banca Examinadora:

- Prof. Dr. Gedson Faria (CPCX/UFMS) - Orientador
- Prof. Me. Deiviston da Silva Aguena (CPCX/UFMS)
- Prof. Bel. Cleiton Gonçalves de Almeida (CPCX/UFMS)

Dedico este trabalho aos meus pais. As duas maiores inspirações de minha vida e que sempre me incentivaram a lutar pelos meus sonhos.

Agradecimentos

A Deus por minha vida, família amigos.

Aos meus pais, minha mãe Ana, meu pai Damião, pelo amor, ensinamentos para a vida e por me apoiarem nas realizações dos meus sonhos.

Meus agradecimentos a minhas irmãs Elaine e Cristiane, sobrinhos Enzo e Lorena e cunhado Hugo por fazerem parte da minha história.

Meus agradecimentos à minha namorada, Ana Larissa pela paciência, carinho e companheirismo quando necessitei.

A todos os professores, por todos os conselhos e ajuda durante os meus estudos, pelo incentivo nos momentos de desânimo e auxílio no desenvolvimento do meu TCC.

Sou grato pelos amigos, irmãos na amizade e colegas que direta ou indiretamente me apoiaram e me deram forças para a conclusão deste trabalho.

Agradecimentos especiais são direcionados ao Prof. Me. Kleber Kruger e ao Prof. Dr. Gedson Faria pela orientação, incentivo confiança.

Enfim, agradeço a todas as pessoas que fizeram parte desta jornada em minha vida.

Quando tudo parecer dar errado em
sua vida, lembre-se que o avião decola
contra o vento, e não a favor dele.

Henry Ford

Resumo

Considerando a vasta possibilidade de utilização de grafos e autômatos em diversas situações e locais a nossa volta, podemos mensurar a importância da implementação de uma ferramenta capaz de auxiliar na modelagem gráfica acerca do assunto. Analisando os dados relativos às matrículas em cursos superiores (INEP - INSTITUTO NACIONAL DE ESTUDOS E PESQUISAS EDUCACIONAIS ANÍSIO TEIXEIRA BRASÍLIA, 2017), fica evidente o quantitativo de pessoas que poderiam utilizar tal ferramenta, visto que as modelagens realizadas nas graduações tradicionalmente são feitas manualmente.

Neste trabalho foi proposto o estudo de tecnologias necessárias ao desenvolvimento de uma ferramenta de auxílio na modelagem de grafos e autômatos, pois as aplicações já existentes apresentam interfaces sem aspectos profissionais, tecnologias desatualizadas e limitações relacionadas à modelagem e manipulação dos grafos ou autômatos.

Meses após o início das implementações da ferramenta proposta, erros inerentes à integração das *threads* da aplicação surgiram, ocasionando a impossibilidade de prosseguir com o projeto. De tal maneira, objetivando contornar os problemas ocorridos ao integrar as API's JavaFX e Java Swing, novas tecnologias foram estudadas e aplicadas ao desenvolvimento da ferramenta. O prazo destinado à implementação da nova versão da ferramenta esgotou-se e assim, sua conclusão não foi possível, pois existe a necessidade de implementação da injeção de dependências entre os componentes da aplicação, ao qual promoveria o funcionamento da ferramenta como um todo.

Sendo assim, até a presente data a *VaucangraphFx* tornou-se um *framework* ao qual sua estrutura ficará disponível para quem deseje implementar sua injeção de dependências ou acoplar funcionalidades à ferramenta. Dependendo das funcionalidades adicionadas à *VaucangraphFx*, pode-se destinar a ferramenta a alunos, professores ou pessoas que façam o uso da plataforma LaTeX para a modelagem de grafos ou autômatos, desde que sejam implementadas as funcionalidades do módulo *vaucangraph-transducer*, que destina-se à tradução do grafo ou autômato modelado na ferramenta para a linguagem do pacote VAUCANSON-G.

Palavras-Chave: Grafo, Autômato, Vaucangraph, Vaucanson-G.

Abstract

Considering the vast possibility of using graphs and automata in various situations and places around us, we can measure the importance of implementing a tool capable of assisting in the graphic modeling of the subject. Analyzing data on enrollment in higher courses (INEP - INSTITUTO NACIONAL DE ESTUDOS E PESQUISAS EDUCACIONAIS ANÍSIO TEIXEIRA BRASÍLIA, 2017), it is evident the amount of people who could use such tool, since the modeling done in the graduations is traditionally done manually.

In this work, it was proposed the study of technologies needed to develop a tool to aid in the modeling of graphs and automata, since the existing applications present interfaces without professional aspects, outdated technologies and limitations related to the modeling and manipulation of graphs or automata.

Months after the implementation of the proposed tool implementations, errors inherent in the application of the applicable laws have arisen, causing the impossibility of proceeding with the project. In this way, in order to overcome the problems that occurred when integrating the APIs of JavaFX and Java Swing, new technologies were studied and applied to the development of the tool. The deadline for the implementation of the new version of the tool was exhausted so its conclusion was not possible, as there is a need to implement the injection of dependencies between the components of the application, which would promote the functioning of the tool as a whole.

Thus, until today, VaucangraphFx has become a framework to which its structure is available to those who wish to implement their dependency injection or to add functionality to the tool. Depending on the functionality added to VaucangraphFx, the tool can be used for students, teachers or people who use the LaTeX platform for modeling of automata or graphs, as long as the functionality of the vaucangraph-transducer module is implemented, which is intended for the translation of the graph or automaton modeled in the tool for the package language VAUCANSON-G.

Keywords: Graph, Automata, Vaucangraph, Vaucanson-G.

Lista de Figuras

Sumário

Capítulo 1

Introdução

Na última década, a procura por cursos superiores têm aumentado de forma significativa, tanto em redes públicas quanto nas redes privadas. Sabendo que estudos relacionados a grafos são realizados em muitos dos cursos ofertados e levando em consideração o elevado número de alunos matriculados, pode-se mensurar o elevado quantitativo que possa necessitar de uma ferramenta de auxílio para modelagem de grafos e/ou autômatos, visto que tradicionalmente tais modelagens são realizadas manualmente.

A teoria dos grafos pode ser utilizada em variadas áreas, tal como foi utilizado por Giraldo, Zuluaga e Espinosa (2014) para demonstrar a utilização de grafos na identificação do *networking* [1] exercido em pequenas empresas, com a finalidade de intensificação de melhorias em pontos observados nessas análises, assim tornando-as competitivas no mercado. Por sua vez, Del-Vecchio et al. (2009) utiliza a teoria dos grafos para identificar padrões no mercado de investimentos, possibilitando que o investidor visualize conjuntos de dados e extraia informações que possam ser úteis aos investimentos.

Algumas ferramentas já foram propostas para a modelagem de grafos, entretanto possuem certas limitações e interfaces pouco atrativas. As ferramentas ROX desenvolvida por Sangiorgi (2006), A-Graph desenvolvida por Lozada (2014) e *Vaucangraph 2.0*, desenvolvida por Souza e Kruger (2010) apresentam interfaces simples, tecnologias ultrapassadas e a necessidade de inserção de funcionalidades que possibilitem uma maior abrangência no estudo e utilização dos grafos modelados.

Neste trabalho propõe-se o estudo de novas tecnologias que possam suprir as necessidades observadas nas ferramentas anteriormente citadas. Por fim, os resultados obtidos serão disponibilizados à comunidade. Os problemas encontrados durante o processo de desenvolvimento deste trabalho também serão expostos afim de evitar que mais pessoas sejam prejudicadas pela falta de informações acerca do assunto.

[1] Networking: indica a capacidade de estabelecer uma rede de contatos ou uma conexão com algo ou com alguém.

1.1 Justificativa

Dados do censo de educação superior do ano de 2017, publicados pelo INEP - Instituto Nacional de Estudos e Pesquisas Educacionais Anísio Teixeira Brasília (2017) evidenciam um crescimento significativo na busca por cursos superiores. Somando as áreas de Ciência da Computação, Matemática e Processamento de Informação, foram realizadas 265.396 matrículas nas redes privadas e públicas. Considerando que ao menos 1/4 desses alunos matriculados estudem a teoria dos grafos ou autômatos, seriam 66.349 pessoas ao ano que poderiam necessitar da ferramenta proposta neste trabalho, visto que normalmente a modelagem de grafos é feita à mão. O crescimento das matrículas em graduações foi de 56,4%, considerando o período de tempo entre os anos de 2007 à 2017, sendo que a média anual foi de 4,6%, podendo presumir o aumento desses indicadores nos próximos anos (INEP - INSTITUTO NACIONAL DE ESTUDOS E PESQUISAS EDUCACIONAIS ANÍSIO TEIXEIRA BRASÍLIA, 2017).

Mesmo existindo aplicações em que propõe-se a modelagem e manipulação de grafos, algumas das tecnologias utilizadas nas implementações destas ferramentas tornaram-se ultrapassadas, podendo gerar instabilidades ou falhas em seu funcionamento. A utilização de novas tecnologias abririam um novo horizonte de possibilidades quanto às funcionalidades da ferramenta, possibilitando a cobertura de falhas ou pontos inexplorados nas demais aplicações. Surge assim, a necessidade de estudos para a implementação de uma ferramenta que possa atender a tais aspectos.

1.2 Objetivos

1.2.1 Objetivo Geral

Tem-se por objetivo principal neste trabalho o estudo de novas tecnologias para a criação de uma aplicação de modelagem de grafos e autômatos.

1.2.2 Objetivos Específicos

São objetivos específicos deste trabalho:

- Estudar a biblioteca *JavaFX* presente no JDK 8.0;
- Estudar o problema e descrever o projeto no padrão MVVM;
- Estudar a biblioteca *mvvmFX* 1.7.0, como ferramenta para auxiliar a manipulação das telas;
- Estudar a biblioteca *JGraphX* 3.9.8, como ferramenta de apoio para a criação de imagens 2D;
- Implementar uma interface interativa para a criação de modelos gráficos de grafos e/ou autômatos;

Contudo, tarefas simples de manipulação de objetos com o mouse, ocasionaram erros de exceção relacionados à integração das API's JavaFX e Java Swing. Para contornar esse problema, propôs-se:

- Estudar a implementação modular do JDK 11;
- Reescrever o projeto no padrão MVVM respeitando a implementação modular;
- Estudar como realizar a remoção da biblioteca *JGraphX* do projeto original;
- Implementar de forma modular uma interface interativa para a criação de modelos gráficos de grafos e/ou autômatos sem a biblioteca *JGraphX*;

1.3 Organização da Proposta

Este trabalho está dividido em cinco capítulos. No capítulo 2 encontra-se a fundamentação teórica, contendo informações básicas sobre grafos, autômatos e sobre a biblioteca JGraph, almejando promover a facilidade na leitura e entendimento deste trabalho.

No capítulo 3 encontra-se todo o desenvolvimento do projeto, descrevendo as tecnologias utilizadas, problemas encontrados e meios utilizados para contornar os problemas evidenciados.

No capítulo 4 são apresentados os testes e resultados adquiridos no decorrer do projeto. No capítulo 5 são feitas as considerações finais do trabalho, bem como sugestões de melhorias que possam ser implementadas em trabalhos futuros.

Capítulo 2

Fundamentação Teórica

2.1 Considerações Iniciais

Neste capítulo serão abordados os conceitos básicos para o bom entendimento e fluidez na leitura deste trabalho. Na Seção 2.2 descreve-se os conceitos e exemplos de utilização de grafos, abordando a diferença entre: grafos direcionados e grafos não direcionados. Por sua vez, a Seção 2.3 relata os conceitos de Autômatos, apontando seus tipos e aspectos de cada um. Por fim, na seção 2.4 apresenta-se a biblioteca *JGraph*, que foi utilizada no desenvolvimento da primeira parte deste trabalho, como descrito na seção 3.2.

2.2 Grafos

Definição 1. *Define-se um grafo por* $\boldsymbol{G = (V, A)}$*, sendo que V representa um conjunto não vazio de vértices (PRESTES, 2016) e A representa um conjunto de arestas, que são associadas a pares não ordenados e não necessariamente distintos de vértices do grafo G (LUCCHESI, 1979).*

Segundo Lucchesi (1979), diagramas podem ser utilizados para representar grafos, sendo que para isto, pontos são utilizados para representar os vértices e suas arestas seriam expressas mediante linhas, que ligariam-se aos seus pares de vértices. Quando dois vértices são ligados por meio de uma aresta, dizemos que nossos vértices são **adjacentes**, por sua vez a aresta é denominada **incidente** aos vértices (JURKIEWICZ, 2009).

Ziviani (2013), exemplifica a internet como um imenso grafo, onde objetos (vértices) representariam os documentos e os *links* simbolizariam as arestas responsáveis pelas conexões de um objeto a outro. Já Feofiloff, Kohayakawa e Wakabayashi (2011), utiliza como exemplo o mapa dos estados do Brasil, sendo que cada vértice simboliza um estado e as arestas que ligam os vértices simbolizam as fronteiras entre estes, como representado na figura 2.1.

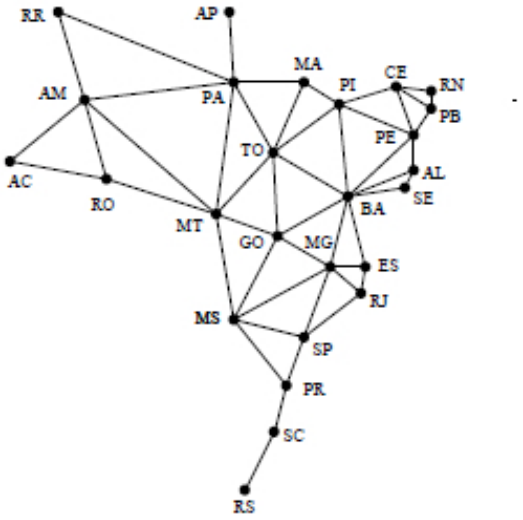


Figura 2.1: Grafo dos estados do Brasil. Retirado de (FEOFILOFF; KOHAYAKAWA; WAKABAYASHI, 2011)

.

De acordo com Feofiloff, Kohayakawa e Wakabayashi (2011), um grafo que não possui arestas paralelas, ou seja, duas arestas diferentes para o mesmo par de vértices e nem possui *laços* [1] é considerado um **grafo simples**. A Figura 2.1 representa um exemplo de grafo simples.

Definição 2. *(JURKIEWICZ, 2009) Um grafo é completo quando todo par de vértices é ligado por uma aresta. Denota-se um grafo completo com n vértices por* K_n.

Na figura 2.2 apresenta-se um exemplo de grafo completo K_6.

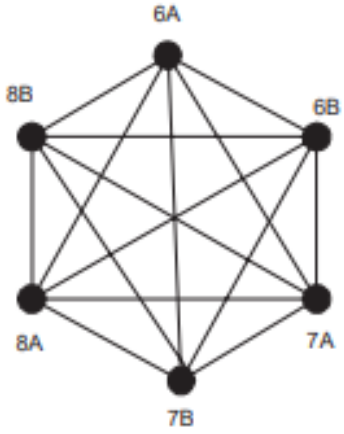


Figura 2.2: Grafo completo K_6. Retirado de (JURKIEWICZ, 2009)

.

[1] Laço: São arestas que ligam vértices a eles mesmos (ZIVIANI, 2007).

Definição 3. *Quando cada aresta de um grafo possui um sentido associado, dizemos que este trata-se de um* ***grafo orientado*** *ou* ***dígrafo*** *(ROSEN, 2009).*

Como forma de exemplificar a utilização de um grafo, Prestes (2016) utiliza o trânsito da cidade como exemplo, sendo que cada entroncamento seria representado mediante um vértice e o fluxo permitido seria a relação entre os entroncamentos, ou seja, seriam as arestas que os ligam. Entretanto, a ida de um determinado entrocamento à outro, não implicaria em sua volta. Sendo assim, existiria a necessidade de identificação destas restrições. De tal maneira, as arestas deste grafo passariam a indicar a direção em que o fluxo aconteceria, tornando assim, o grafo orientado (PRESTES, 2016). A figura 2.3 representa um grafo orientado simples.

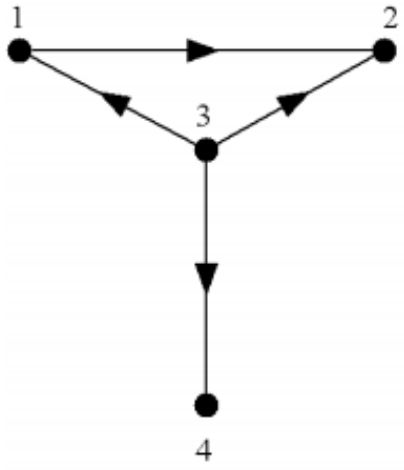


Figura 2.3: Grafo orientado simples. Retirado de (PRESTES, 2016)

.

Definição 4. *O tamanho de um grafo G é definido pela soma do total de seus vértices com o total de suas arestas. Quando um grafo não possui vértices e nem arestas denomina-se* ***grafo vazio*** *(LUCCHESI, 1979).*

2.3 Autômatos

Definição 5. *(MENEZES, 2009) Um* ***autômato finito determinístico*** *ou apenas* ***autômato finito*** *é definido por uma quíntupla ordenada, que apresenta-se da seguinte maneira :*

$$\boldsymbol{M} = (\ \Sigma\ ,\ Q,\ \delta,\ q_0,\ F\ )$$

Sendo que:

Σ *: é o alfabeto de entrada.*

Q*: é um conjunto finito de estados possíveis do autômato.*

δ *: é a função de transição.*

q_0 : é um elemento de Q que se distingue dos demais e que denomina-se estado inicial.

F : trata-se de um subconjunto de Q, que são denominados estados finais.

Autômatos são utilizados para análise de dados provenientes de uma fita de entrada entretanto, não geram uma saída, pois sua funcionalidade é apenas indicar se a entrada é aceitável ou não para a linguagem determinada (LEWIS; PAPADIMITRIOU, 2000).

Segundo Santos et al. (2017) autômatos são utilizados em diversas áreas, podendo ser utilizados desde os mais simples processos até os mais complexos.

Na imagem 2.4 um diagrama é utilizado para representar as transições entre os estados de um autômato.

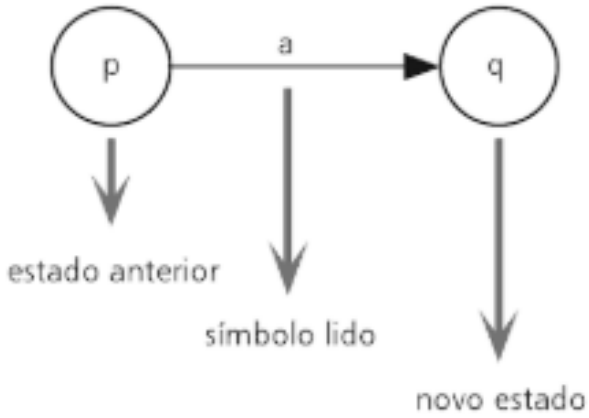


Figura 2.4: Diagrama da transição entre estados de um Autômato finito. Retirado de (MENEZES, 2009)

.

Conforme Menezes (2009), após o processamento de qualquer entrada o autômato finito para, visto que toda palavra é finita e assim um novo símbolo será lido, excluindo de tal maneira a possibilidade de um ciclo infinito.

2.4 JGraph

A representação de dados graficamente é imprescindível nas mais variadas áreas e disciplinas, entretanto, mesmo com a existência de diversas ferramentas que dão suporte à criação de grafos, a maioria apresenta limitações, sejam elas ocasionadas por incompatibilidade do sistema operacional, por requisitarem formatos específicos de dados ou pela limitação do formato de saída, restringindo desta forma, seu uso a determinados ambientes de processamento textual (PLANK, 1993).

Um objeto *JGraph* não contém os dados propriamente ditos, ele apenas fornece uma visão das informações contidas no modelo de dados, assim como qualquer componente *Swing* (ALDER, 2001).

Devido à mudança de arquitetura após sua 5ª versão, a biblioteca *JGraph* passou a utilizar o nome de *JGraphX*, iniciando agora a partir da versão 1.x, mas pode-se considerar como sendo sua 6ª versão (JGRAPH LTD, 2019).

Segundo Plank (1993) a biblioteca *JGraph* pode ser utilizada em todas as plataformas disponíveis, não restringindo seu uso a terminais ou sistemas operacionais específicos, além de ser bem documentado e gratuito.

Segundo JGraph Ltd (2019) a habilidade do desenvolvedor e da plataforma *Swing* são os fatores que limitarão a modelagem e exibição das células modeladas, visto que a biblioteca possui diversos recursos para tal. Existem as possibilidades de os vértices serem personalizados, fazendo a utilização de formas, desenhos vetoriais, imagens, entre outras possibilidades.

Capítulo 3

Desenvolvimento

3.1 Considerações Iniciais

Neste capítulo apresentam-se técnicas e metodologias utilizadas na implementação da ferramenta proposta neste trabalho. Este capítulo teve seu escopo dividido em duas partes sendo que, em sua primeira parte apresenta-se a etapa inicial de desenvolvimento da ferramenta, até o momento em que a ferramenta tornou-se inexequível, como apresentado na seção 3.2. Visando contornar os problemas que impossibilitaram a continuidade deste projeto, novas tecnologias foram estudadas e sua utilização e aplicação serão abordadas na segunda parte deste capítulo, que se inicia na seção 3.3 deste trabalho.

3.2 Vaucangraph 3.0

Ao entrar em contato com aplicações que visam auxiliar no aprendizado de grafos e autômatos, surgiu o entusiasmo em explorar tecnologias que possam viabilizar tal feito. O contato com a aplicação *Vaucangraph 2.0* (SOUZA; KRUGER, 2010) resultou no encorajamento para buscar formas de desenvolvimento de uma nova ferramenta utilizando tecnologias atualizadas, objetivando melhorias na utilização dos recursos oferecidos à aplicação, melhor desempenho no processamento das informações e melhorias quanto à apresentação visual da ferramenta.

Tendo em mente quais tecnologias e ferramentas possivelmente seriam empregadas, uma rotina de estudos iniciou-se buscando por aplicações semelhantes à almejada, pois só assim seria possível mensurar a grandeza e complexidade a serem empregados no projeto, permitindo de tal modo a melhoria de alguns pontos fracos em relação à ideia inicial e em relação às aplicações concorrentes.

Após levantar os requisitos necessários ao projeto, obteve-se uma estrutura que serviria como base ao longo da implementação, tendo em mente que a qualquer momento poderiam surgir imprevistos, novos requisitos ou funcionalidades a serem incorporados ao projeto. Sendo assim, estudos do Java 8 foram iniciados buscando ampliar os conhecimentos acerca da versão, que por sua vez já havia sido utilizada durante a graduação, porém algumas

dependências existentes no projeto não foram abordadas de forma aprofundada quanto o necessário. A biblioteca *JavaFX*, presente no java 8 ganharia grande foco no momento, pois tornava-se imprescindível a criação de *views* [1] baseadas no *UX Design*, que proporcionariam o fácil manuseio e entendimento do usuário, minimizando assim a necessidade de conhecimento técnico acerca da ferramenta em desenvolvimento e aprimorando o tempo gasto para realização das tarefas desejadas (TEIXEIRA, 2014).

Até então conseguia-se produzir *views* básicas com cores e estilos padrões do sistema operacional, entretanto eram pouco atrativas. Tal fato ocasionou a necessidade do uso de CSS para melhoria de tais aspectos. Buscando facilitar o desenvolvimento das telas da aplicação, utilizou-se a IDE *JavaFX Scene Builder 2.0*, que proporcionou otimização e agilidade no desenvolvimento gráfico do projeto.

Com inspiração na técnica dividir e conquistar, a aplicação foi fragmentada em pequenos "problemas" e suas funcionalidades foram implementadas uma por vez, almejando a redução da complexidade e visando o estudo do que fosse necessário no momento, assim quando todos os "problemas" propostos fossem solucionados, o código seria reagrupado e resultaria na aplicação final.

Almejando ampliar a integridade e operabilidade da aplicação, cada componente da ferramenta seria desenvolvido e testado individualmente e por fim, unidos para dar origem ao *software* [2].

3.2.1 Divisão da aplicação

Como forma de organizar o processo de implementação do *Vaucangraph 3.0*, elencou-se as funcionalidades a serem inseridas na aplicação. De tal maneira, foram realizadas as divisões dos componentes, de forma em que apenas as classes necessárias estariam agrupadas em cada pacote. Por fim, ao executar a aplicação cada componente seria inicializado e incluído em uma classe única, dando origem à janela principal da ferramenta.

A aplicação foi dividida em sete componentes, sendo eles:

- Barra de Menu.
- Barra de Ferramentas.
- Paleta de ferramentas.
- Painel de modelagem.
- Painel de inspeção.
- Painel de Propriedades.
- Barra de Status.

[1] **View :** tela ou layout.

[2] **Software:** todo programa armazenado em discos ou circuitos integrados de computador, esp. destinado a uso com equipamento audiovisual.

O foco principal neste momento daria-se ao desenvolvimento do componente “Painel de modelagem”, tanto por seu grau elevado de complexidade quanto por sua importância no projeto.

Buscando por uma biblioteca que proporcionasse facilidade e elegância na modelagem e manipulação de grafos e autômatos, a versão Java da biblioteca *JGraph* foi selecionada, levando em consideração a popularidade na comunidade desenvolvedora, atualizações periódicas e eficiência na solução dos problemas propostos neste trabalho. Perguntas realizadas no forum específico da biblioteca atingiram o total de 27.300 visualizações, como evidenciado na figura 3.1. A alta busca por auxílio em relação à biblioteca *JGraph* confirma sua popularidade e grande utilização como meio de modelagem de grafos e autômatos.

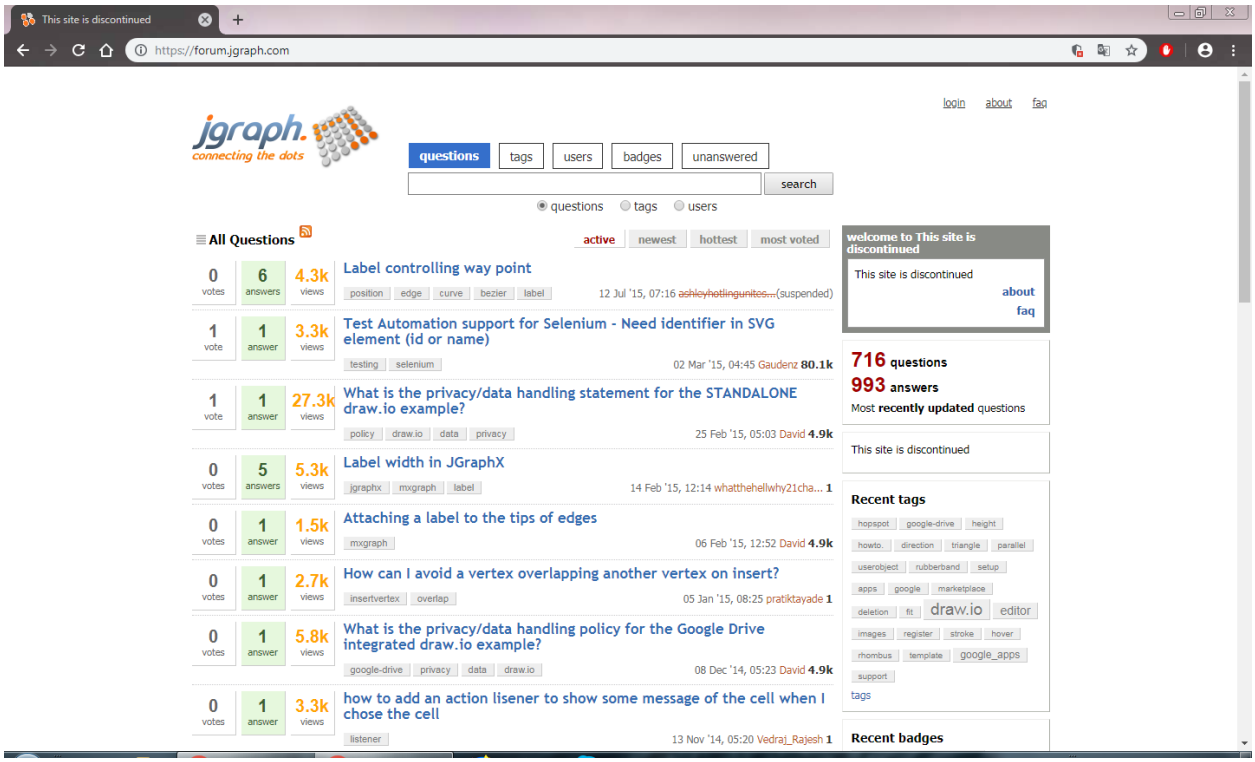


Figura 3.1: Forum destinado ao uso da biblioteca JGraph.

Tal forum foi descontinuado e atualmente dúvidas relacionadas às versões atuais da biblioteca são solucionadas na comunidade on-line “Stack Overflow”, que visa ajudar qualquer pessoa que busque auxílio em seu aprendizado, partilhar conhecimentos ou moldar sua carreira.

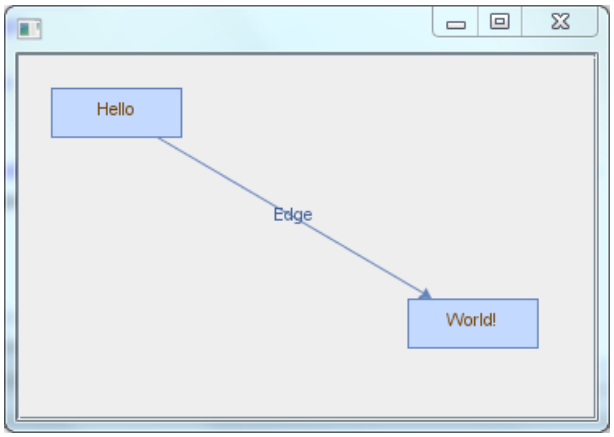
Após definir que a biblioteca *JGraphX* seria utilizada neste trabalho, estudos acerca de seu funcionamento foram iniciados, almejando o domínio quanto à modelagem gráfica utilizando suas funcionalidades. Sendo assim, alguns dos exemplos contidos em sua documentação foram implementados, auxiliando no entendimento de como funcionaria a biblioteca. Tais exemplos foram reproduzidos utilizando as classes do *Java Swing*, como por exemplo o *JFrame*, que é utilizado para a construção de uma tela e que teria o conteúdo da *JGraphX* incluído em seu escopo.

Entretanto, a aplicação seria implementada utilizando a biblioteca *JavaFX*, devido ao

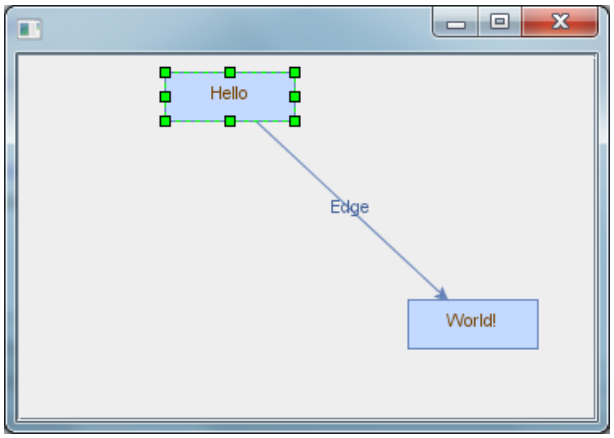
seu maior poder de criação, manipulação e estilização que o *Java Swing*. Deste modo, o novo desafio seria integrar as telas criadas em *JavaFX* e a biblioteca *JGraphX*, desenvolvida em *Java Swing*, para que interagissem entre si, visto que seriam tecnologias distintas.

Decidiu-se então utilizar um componente do *JavaFX* chamado *SwingNode*, que destina-se à adaptação e integração entre API's e *Java Swing*. Graças a isso, a biblioteca *JGraphX* pôde ser integrada ao Painel de Modelagem e então prosseguiu-se para a fase de testes.

Inicialmente os testes do componente consistiam basicamente na inserção de dois vértices, uma aresta e na manipulação destes, como representado na figura 3.2.



(a) estado inicial



(b) estado após evento de clique e arraste.

Figura 3.2: Teste de manipulação de vértices.
Fonte:Elaborada pelo autor.

As API's *JavaFX* e *Java Swing* são executadas em plataformas de threads distintas e específicas de cada API sendo que, tais threads são responsáveis pelo monitoramento e despacho dos eventos para a GUI. Entretanto, devido a complexidade de implementação das API's utilizadas neste trabalho, bugs provenientes dos eventos de *Drag and Drop* foram encontrados e reportados pela comunidade, porém a falta de continuação da biblioteca *Swing* pode ter ocasionado a falta de esforços para corrigir tal problema. O erro desencadeado pode ser visto na figura 3.3

Figura 3.3: Erro ao executar os eventos de Drag and Drop.

Diversos desenvolvedores questionaram a presença do erro em foruns, entretanto soluções não foram encontradas e aguardava-se que a solução viesse em futuras atualizações do Java.

A percepção do problema foi tardio e as buscas por soluções eram constantes, mas as informações encontradas eram limitadas e pouco esclarecedoras.

O tempo para entrega deste trabalho esgotou-se e visto que a aplicação não poderia ser finalizada com as tecnologias utilizadas no momento, a adoção de novas formas de implementação da ferramenta foram levadas em consideração e optou-se pela reescrita do projeto com novas tecnologias.

Assim sendo, a utilização do componente *Webview* foi estudada, baseando-se na integração entre o *Java* com a versão *JavaScript* da biblioteca *JGraph*, conhecida como *mxGraph* porém, a integração de tecnologias *desktop* e tecnologias *web* elevariam o grau de dependência e complexidade do projeto, tornando futuras atualizações custosas. Perante isso, a busca por novos meios continuaram, almejando o desenvolvimento de uma aplicação com o menor número de bibliotecas externas possíveis.

3.3 VaucangraphFx

Sabendo que o projeto anteriormente definido seria inexequível, iniciaram-se os estudos buscando formas de reescrever a estrutura da aplicação, de modo que o uso de bibliotecas e API's externas fossem minimizadas, reduzindo assim as dependências do projeto.

Nesse período de tempo a 11ª versão do Java foi lançada, despertando o interesse pela programação modular, que proporcionaria facilidade em futuras atualizações da aplicação proposta. Por ter perdido muito tempo com os erros de integração das tecnologias citadas na seção 3.2 deste trabalho, o estudo da remoção do uso da biblioteca *JGraphX* foi feito, visando contornar a reincidência de erros na aplicação.

Segundo Britch (2017) a boa organização e separação das camadas do padrão de desenvolvimento *Model-View-ViewModel* (MVVM), auxiliam na resolução de problemas de desenvolvimento e podem tornar os testes, manutenções e atualizações mais fáceis. Desta maneira, o padrão de desenvolvimento adotado neste trabalho continuou sendo o *Model-*

View-ViewModel **(MVVM)**. Foi feito também o uso do *framework mvvmFX*, sabendo que sua utilização destina-se à implementação com padrão de desenvolvimento *MVVM* e com o *JavaFX* (CASALL; MAUKY, 2017).

Conforme Microsoft (2019) o padrão de desenvolvimento selecionado é organizado da seguinte maneira:

- ***Model***: A lógica de negócio necessária ao funcionamento da aplicação é desenvolvida nesta camada, não tendo relação com o que é apresentado ao usuário. Deste modo, qualquer lógica de negócios ou dados fazem parte da camada *Model* .
- ***View***: Esta camada é responsável pela definição da estrutura, do layout e da aparência do que é exibido ao usuário (BRITCH, 2017).
- ***ViewModel***: Em aplicações desenvolvidas com o presente padrão de desenvolvimento, a camada *ViewModel* é responsável por interligar as operações entre a camada *View* e a camada *Model*. As operações que o usuário pode executar na *view* são definidas nesta camada, bem como as propriedades às quais a camada *view* liga-se.

Na figura 3.4 a divisão e relação entre as camadas do padrão MVVM são apresentadas, sendo que demonstra-se a ligação entre a *view* e a *ViewModel* mediante *data binding*, que são vinculações de dados. Também demonstra-se as atualizações enviadas da camada *ViewModel* à camada *Model*, bem como as notificações enviadas entre elas.

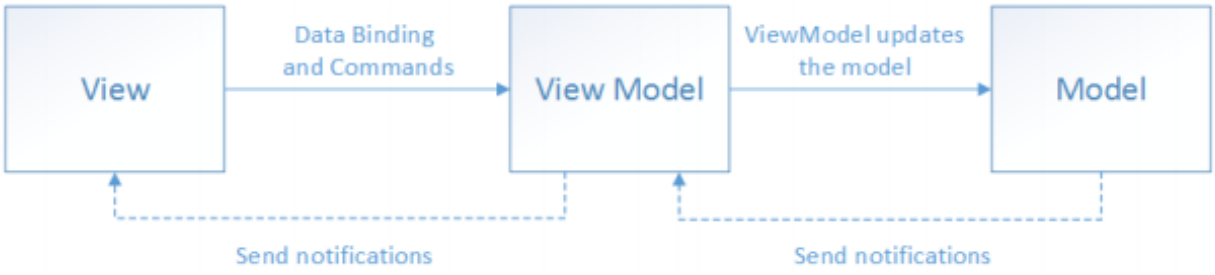


Figura 3.4: Relação entre as camadas do padrão MVVM. Retirado de (BRITCH, 2017)

.

Com a mudança da estrutura da ferramenta para a programação modular, houve a necessidade de reescrita das implementações relacionadas à manipulação das telas da ferramenta.

Segundo Miller, Vandome e McBrewster (2010) a ferramenta *Apache Maven* é utilizada como forma de gerenciar as dependências existentes no projeto e automatizar seus *builds*[3]. Portanto, considerando os possíveis benefícios proporcionados ao projeto, a ferramenta *Apache Maven* foi utilizada.

Dando continuidade ao estudo da estrutura a ser desenvolvida neste trabalho, observou-se as funcionalidades e propósitos que cada componente e classe a ser desenvolvida tinham em comum. De tal maneira, definiu-se a qual módulo seriam destinados, sendo eles:

[3]**Build :** Construção da aplicação

vaucangraph-app, ***vaucangraph-components***, ***vaucangraph-core***, ***vaucangraph-drawing*** e ***vaucangraph-transducer***.

A figura 3.5 demonstra como foi realizada a divisão dos módulos.

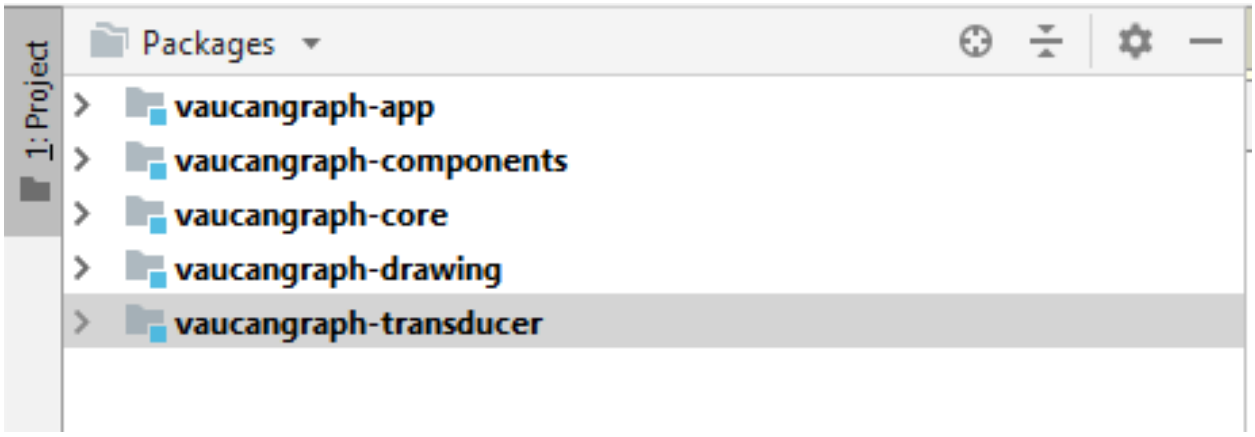


Figura 3.5: Vaucangraph - módulos da aplicação.

A responsabilidade de cada módulo apresenta-se da seguinte forma:

- ***vaucangraph-app*** : Neste módulo encontra-se a classe principal da *VaucangraphFx*, denominada *VaucangraphApp*, que destina-se a iniciar a aplicação.
- ***vaucangraph-components***: Todos os componentes contidos na aplicação, bem como suas funcionalidades encontram-se neste módulo. Cada componente aqui desenvolvido possui um *FXML* próprio para gerar suas *views* e que são incluídos posteriormente em um único *FXML* denominado *Main*, que é responsável por gerar a estrutura principal da aplicação.
- ***vaucangraph-core*** : O módulo *Core* contém todas as classes fundamentais ao armazenamento dos dados de toda a aplicação, sendo que todos os outros módulos ligam-se à este via *Data Binding*. Deste modo, todas as alterações realizadas na ferramenta seriam interligadas, garantindo sempre dados atualizados nas classes implementadas no *vaucangraph-core* e assim consequentemente, nos demais módulos ligados à este. Este módulo recebeu tal nome por tratar-se do coração da ferramenta.
- ***vaucangraph-drawing*** : Com a decisão de remoção da biblioteca *JGraphX* da aplicação e desejo de reduzir as dependências de bibliotecas e *API's* externas, foram necessárias as implementações de classes destinadas à representação visual dos grafos e/ou autômatos. Sendo assim, este módulo tem como objetivo abrigar todas as classes responsáveis por calcular ângulos, distâncias e tamanhos dos vértices e arestas, bem como a estilização dos grafos a serem apresentados ao usuário final.
- ***vaucangraph-transducer*** : Este módulo é responsável por abrigar as classes destinadas à captura de dados do subcomponente responsável por representar visualmente grafos ou autômatos modelados, sendo estes convertidos para comandos do pacote VAUCANSON-G, aos quais seriam destinados à representação de grafos e/ou autômatos

em documentos produzidos na plataforma LaTeX. Toda a transdução de linguagem seria realizada de forma automática e os dados finais alimentariam o subcomponente contido no *Painel de Modelagem*, localizado no módulo *vaucangraph-components*.

Inicialmente foram moldadas as classes responsáveis pelo armazenamento dos dados da aplicação, sendo estas destinadas ao módulo *vaucangraph-core*, responsável por alimentar os dados dos demais módulos ligados ao módulo *vaucangraph-core*.

Para modelagem de um grafo ou autômato o usuário precisaria criar um projeto, onde seriam fornecidas informações básicas para o bom funcionamento da aplicação. Tais informações seriam basicamente: o tamanho do painel de modelagem, tipo de projeto a ser criado (grafo, dígrafo ou autômato) e nome do projeto. Sendo assim, poderia-se inicializar a tela principal da aplicação com os dados, ícones e componentes ajustados da forma ideal. Tais informações inseridas ao iniciar um novo projeto seriam armazenadas nas classes do módulo *vaucangraph-core*, sendo elas fornecidas aos componentes que necessitem destas informações e que estejam vinculados à este módulo.

Como exemplo de vinculação de dados realizado na *VaucangraphFx*, temos o componente *Inspector*, que liga-se ao módulo *vaucangraph-core* e utiliza as informações do projeto necessárias ao seu funcionamento, podendo assim criar sua árvore de visualização, que atualiza-se de forma automática perante alterações realizadas.

Outro exemplo de vinculação de dados pode ser observado no componente denominado "barra de ferramentas", que contém campos que informam o tamanho do painel de modelagem e que possibilitam a sua alteração após criar o projeto. Devido à sua vinculação bidirecional, quaisquer alterações efetuadas nos dados de um componente, implicam na atualização dos dados armazenados pelas classes do módulo *vaucangraph-core* e vice-versa.

Como grande parte da aplicação envolve a modelagem gráfica de grafos e autômatos e visto que os problemas anteriormente citados na seção 3.2 deste trabalho forçaram a mudança das tecnologias utilizadas, priorizou-se a busca por soluções quanto à remoção da biblioteca *JGraphX* da estrutura da aplicação.

3.3.1 Remoção da biblioteca JGraphX

A remoção da biblioteca *JGraphX* reduziria a quantidade de dependências do projeto, além de contornar os problemas ocasionados por sua integração com a *JavaFX* mediante utilização do componente *SwingNode*.

Para tal feito, optou-se pela implementação de forma nativa das classes destinadas a gerar gráficos 2D para representação dos grafos e autômatos. As classes implementadas com tal objetivo foram destinadas ao módulo *vaucangraph-drawing*.

Visando a representação gráfica dos vértices, foi feita a utilização do componente *Circle* do *JavaFX*. Como forma de representação das arestas, foram utilizados os componentes *Line*, em caso de arestas não-orientadas, e no caso de arestas orientadas foi feita a utilização de forma combinada dos componentes *Line* e *Polygon*, sendo estes reunidos em um *Group* herdado por sua classe de modelagem.

Na modelagem gráfica das arestas em arco, utilizou-se os componentes *ArcTo* e *MoveTo*,

sendo que utilizariam os vértices de origem e de destino, respectivamente, como extremidades do arco. Por fim, os dois seriam unidos em um componente *Path* e posteriormente incluídos ao *Group* herdado em sua classe de modelagem.

Pelo fato de as arestas ligarem-se sempre ao centro dos vértices, foram necessários os cálculos da distância e do ângulo entre dois pontos. Desta forma, com o resultado destes cálculos, as extremidades das arestas poderiam ser ligadas da forma correta à borda dos vértices e toda vez que um destes fossem movidos, os cálculos seriam realizados novamente, atualizando em tempo real a posição à qual as arestas deveriam ligar-se.

As demais classes implementadas para a modelagem gráfica em 2D destinam-se à manipulação dos dados dos grafos ou autômatos modelados, que serão armazenados pelas classes do módulo *vaucangraph-core*. Desta maneira, as alterações feitas nos grafos moldados implicam nas alterações automáticas em todos os módulos e componentes vinculados aos dados contidos no módulo *vaucangraph-core* via *data binding*.

Após o desenvolvimento das classes necessárias ao desenvolvimento gráfico em 2D básico dos grafos, prosseguiu-se para o desenvolvimento dos demais componentes da aplicação.

3.3.2 Implementação dos Componentes

Ao iniciar as implementações das funcionalidades dos componentes da aplicação, contidos no módulo *vaucangraph-components*, o prazo destinado à implementação da ferramenta encontrava-se próximo ao fim. De tal maneira, o desenvolvimento das funcionalidades inerentes aos componentes era intenso, buscando o máximo aproveitamento possível de tal período de tempo.

Como os arquivos *FXML* necessários às *views* já tinham sido modelados na fase de implementação descrito na seção 3.2, apenas adequações à atual estrutura a ser seguida foram feitas, além das mudanças visuais.

Neste momento, a aplicação encontrava-se fracionada em pequenos componentes, buscando manter o foco naquilo que se implementaria no momento. Assim, foi selecionado de forma aleatória um dos componentes, o qual decidiu-se implementar primeiro.

A implementação das funcionalidades iniciariam-se com o componente *inspector*. A dificuldade encontrada ao implementar tal componente foi relacionado à como apresentar de forma agradável, objetiva e intuitiva, os dados de cada vértice e aresta contido no grafo. No período de testes, apenas textos foram utilizados para indicar os vértices de origem, destino e aresta incidente nestes, entretanto aprimorou-se tal representação e o uso de ícones foi adotado.

Logo após, os demais componentes começaram a ser desenvolvidos, sempre com o intuito de aprimorar a utilização dos recursos que as tecnologias utilizadas ofereciam.

3.3.3 Diagrama de dependência de módulo

Nesta seção foram definidas as formas que a ferramenta *IntelliJ IDEA* utilizou para representação nos diagramas de dependências de módulo. Sendo assim, apenas as relações entre

os módulos que não sejam do sistema, tampouco módulos automáticos serão evidenciados. Portanto, apenas as dependências entre os módulos ***vaucangraph-app***, ***vaucangraph-components***, ***vaucangraph-core***, ***vaucangraph-drawing*** e ***vaucangraph-transducer*** serão abordados.

Como melhor forma de exemplificar as dependências entre os módulos do projeto, diagramas gerados na ferramenta *Intellij IDEA* serão utilizados.

Para demonstrar determinadas conexões entre módulos e dependências, utiliza-se as seguintes cores (INTELLIJ IDEA, 2019):

- Setas azuis : Reservam-se as setas azuis para representar as dependências de bibliotecas e módulos.
- Setas verdes : Utiliza-se a cor verde para representar dependências de teste.
- Setas vermelhas : Utiliza-se a cor vermelha para indicar dependências circulares.

A figura 3.6 apresenta as dependências do módulo *vaucangraph-app*. A imagem expressa a dependência correlacionada dos módulos da aplicação, já que o módulo *vaucangraph-app* é destinado à inicialização da aplicação e que necessita de informações contidas no módulo *vaucangraph-components* para que isso ocorra corretamente. Por sua vez, os componentes implementados no módulo *vaucangraph-components* são alimentados com dados provenientes do módulo *vaucangraph-core*.

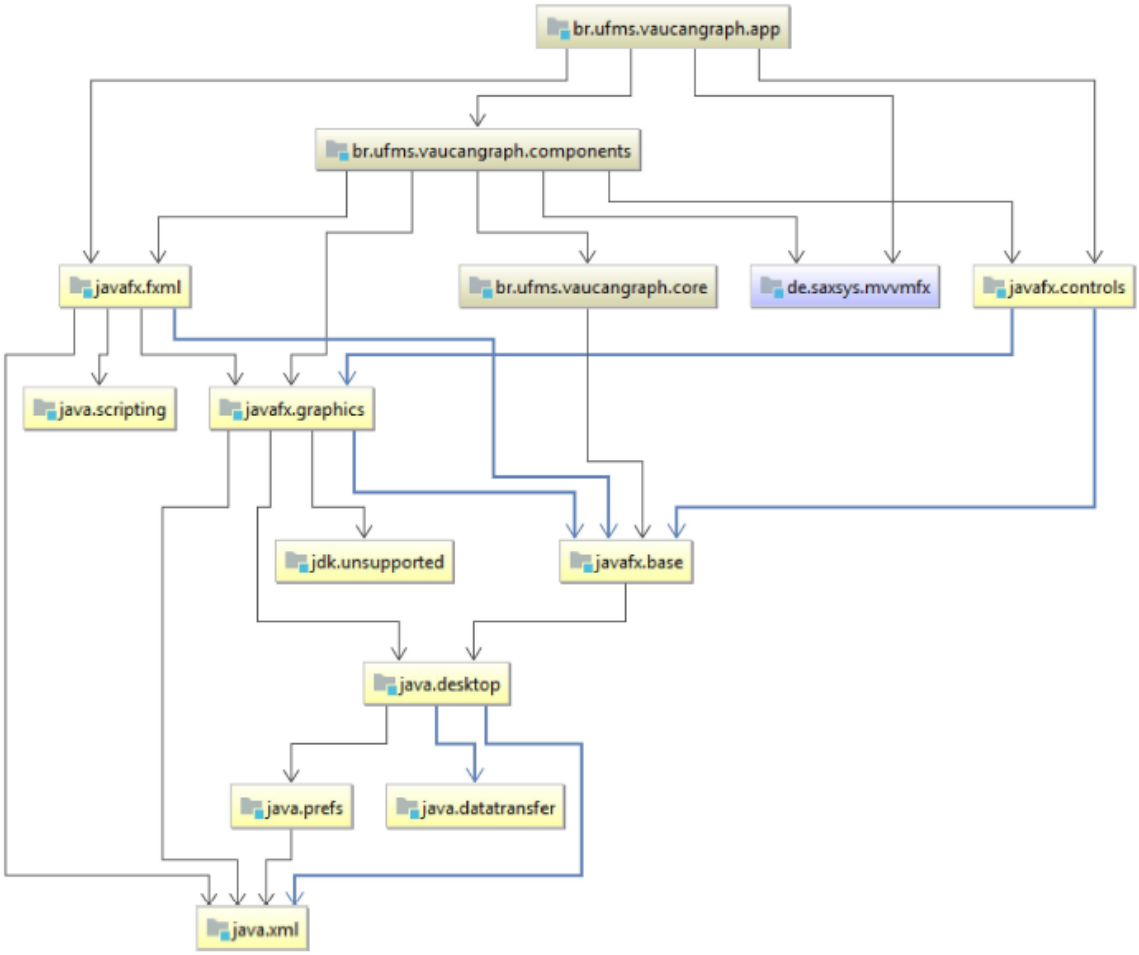


Figura 3.6: Diagrama das dependências do módulo *vaucangraph-app*.

A figura 3.7 demonstra a independência do módulo *vaucangraph-core* quanto à outros módulos, já os demais ligam-se a este para obtenção de dados e o inverso não ocorre.

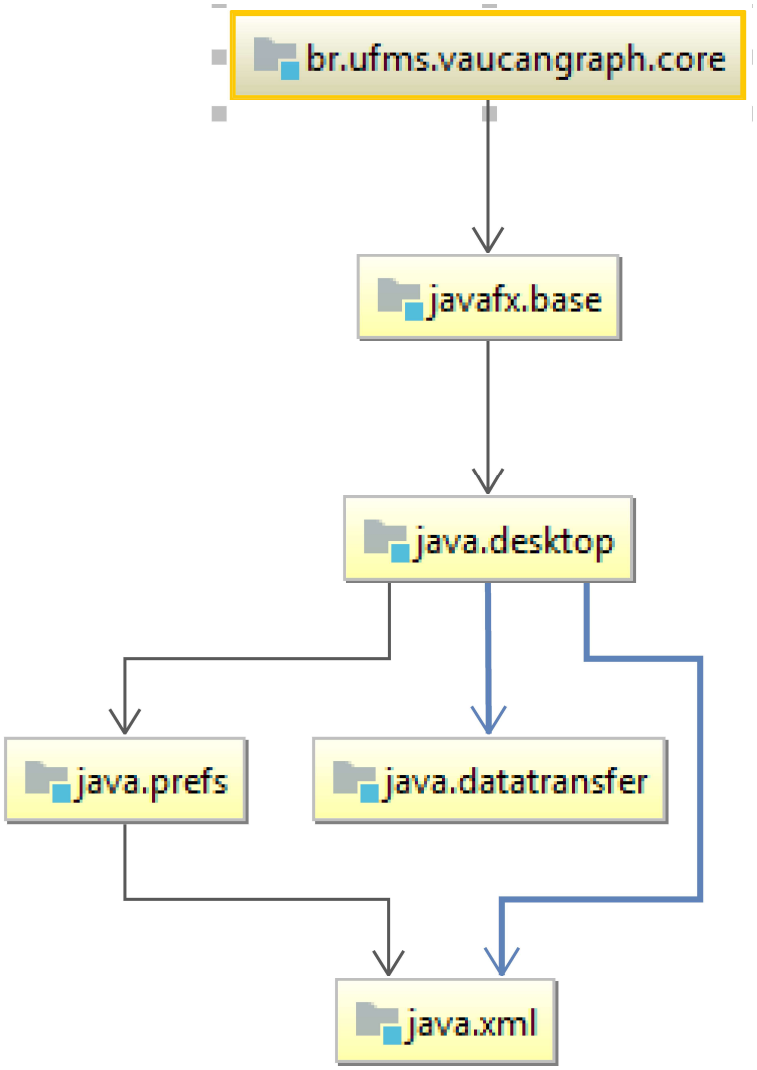


Figura 3.7: Diagrama das dependências do módulo *vaucangraph-core*.

Como existe a necessidade de que os dados dos grafos ou autômatos modelados sejam armazenados nas estruturas das classes desenvolvidas no módulo *vaucangraph-core* de forma correta, a dependência de tal módulo torna-se indispensável, como apresentado na figura 3.8.

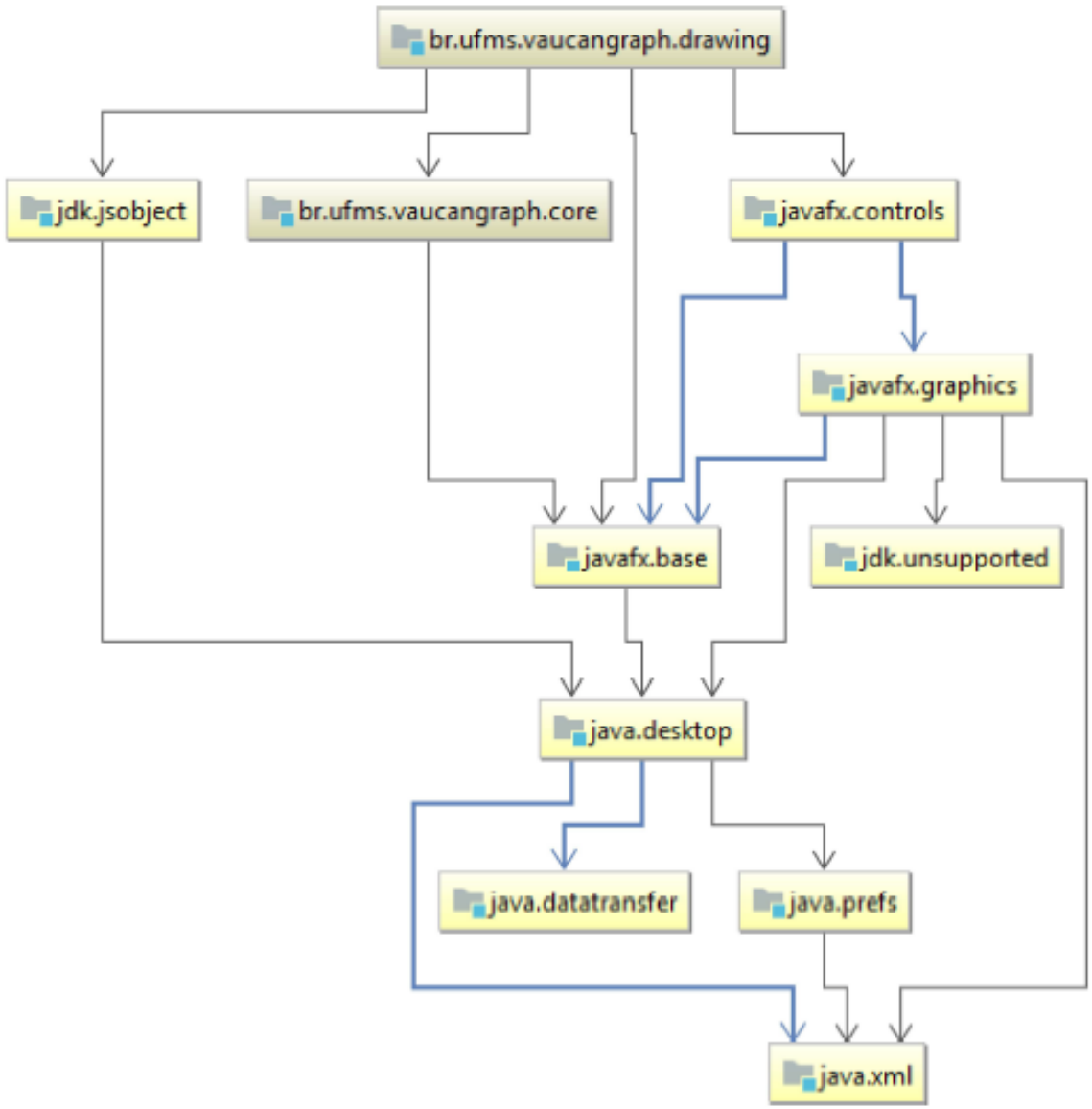


Figura 3.8: Diagrama das dependências do módulo *vaucangraph-drawing*.

Por fim, a figura 3.9 apresenta as dependências do módulo *vaucangraph-transducer*. Este módulo depende de informações, que seriam armazenadas no módulo *vaucangraph-core*, dos grafos ou autômatos modelados. Com tais dados, a tradução para a linguagem do pacote VAUCANSON-Gpoderia ser realizada.

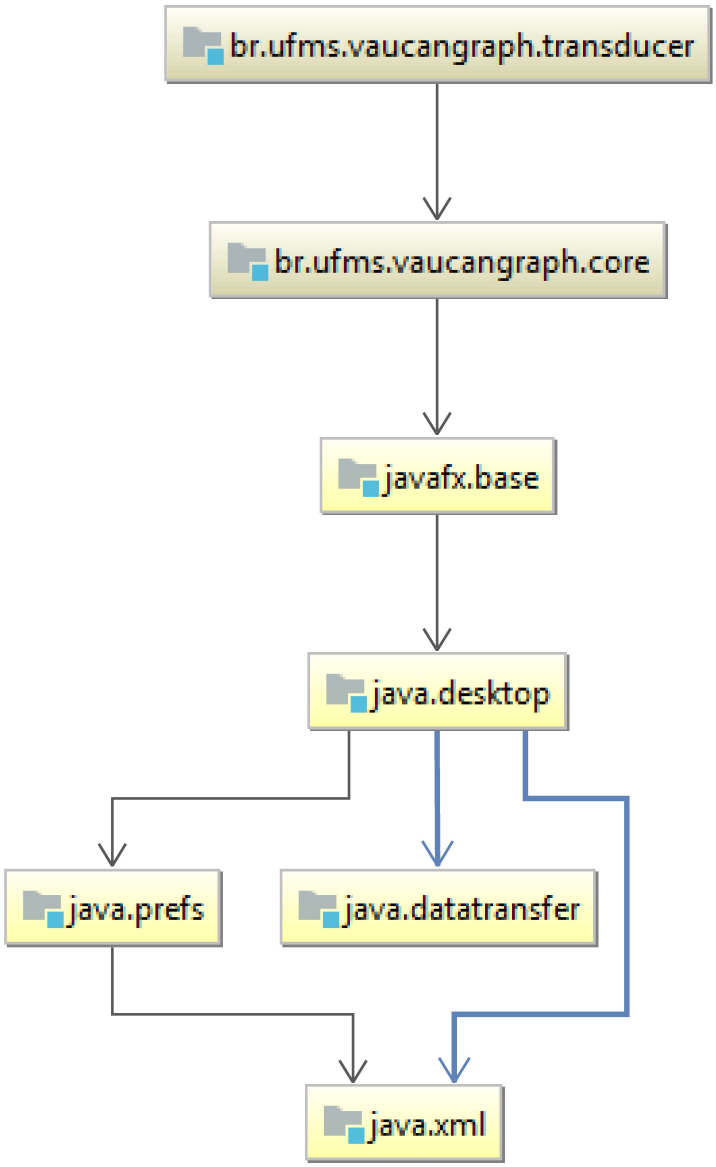


Figura 3.9: Diagrama das dependências do módulo *vaucangraph-transducer*.

Capítulo 4

Resultados

Neste capítulo apresenta-se os resultados provenientes da implementação do *Vaucan-graphFx*. Existindo a necessidade da conclusão da injeção de dependências para que a aplicação funcione como um todo, alguns componentes não puderam ser finalizados, visto que o prazo para implementação da ferramenta esgotou-se. Entretanto, os objetivos propostos neste trabalho foram atingidos. Sendo assim, os resultados obtidos são evidenciados neste capítulo.

4.1 Classes para desenho de gráficos 2D

Após a remoção da biblioteca *JGraphX* da ferramenta, observou-se a necessidade de um meio de modelagem gráfica dos grafos e autômatos. Entretanto, almejava-se a redução de dependências na aplicação, tornando-a mais independente, de tal forma, iniciaram-se as implementações das classes necessárias para modelagem gráfica em 2D dos grafos, dando desta forma, continuidade à aplicação. Tomou-se como objetivo inicial a modelagem de grafos não-direcionados, visto que a complexidade de representação gráfica seria menor que dos grafos direcionados.

Neste momento, a dificuldade daria-se ao fato de as arestas ligarem-se sempre ao centro de cada vértice, gerando resultados não satisfatórios. Deste modo, surgiu a necessidade de utilização do cálculo da distância entre dois pontos, almejando encontrar o ponto ao qual a aresta ligaria-se à borda do vértice. Tal cálculo seria imprescindível, visto que seria utilizado sempre que o ângulo dos vértices ligados pela aresta mudassem.

Após o término dos cálculos e implementações necessárias à estética e boa apresentação dos vértices e arestas modelados, obteve-se como resultado o apresentado na imagem 4.1.

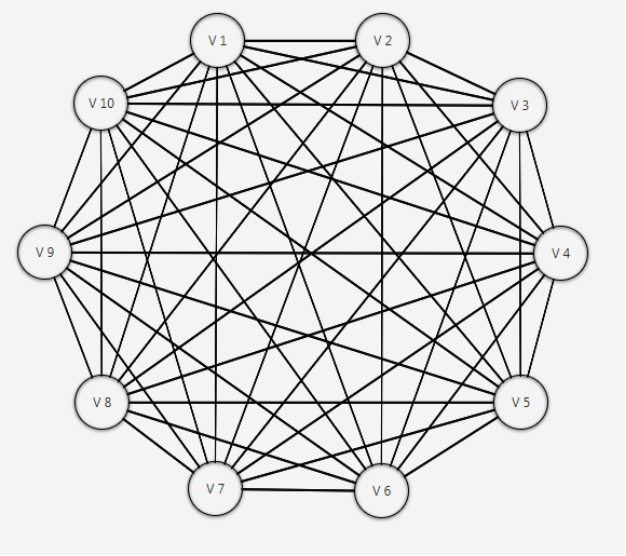


Figura 4.1: Teste das classes desenvolvidas para modelagem de gráficos 2D de grafos - Grafo não-direcionado.

Prosseguindo com os testes, arestas curvas foram utilizadas para conectar os pares de vértices. O resultado é representado na figura 4.2.

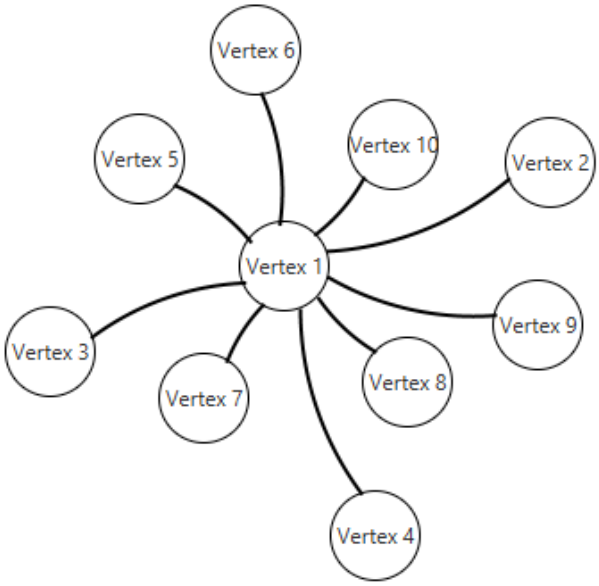


Figura 4.2: Teste das classes desenvolvidas para modelagem de gráficos 2D de grafos - Grafo não-direcionado com arestas curvas.

Após tal feito, as implementações almejando grafos direcionados iniciaram-se após dias de buscas e implementações quanto à inserção de uma seta na ponta da aresta. Como teste, foi modelado um grafo completo K_{10}, que é apresentado na figura 4.3.

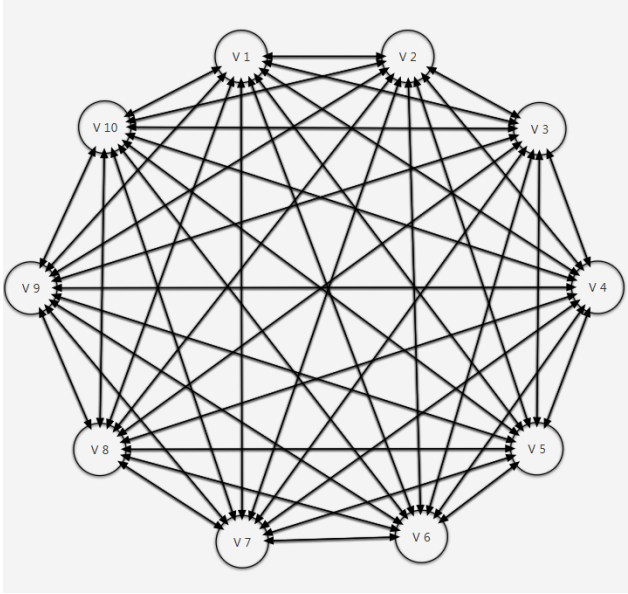
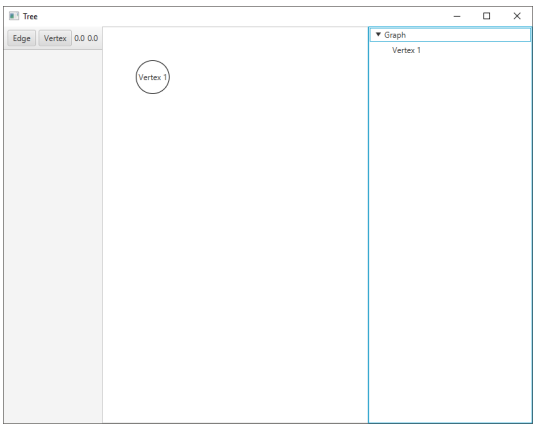


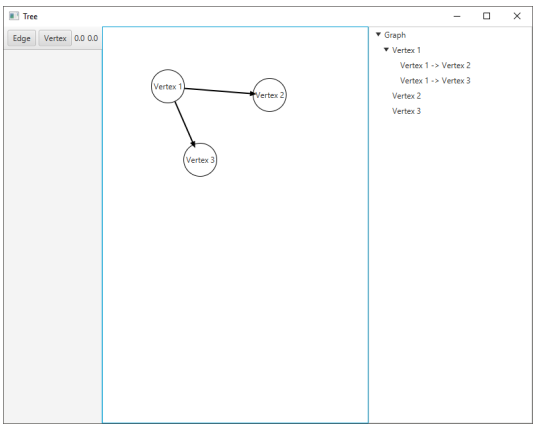
Figura 4.3: Teste das classes desenvolvidas para modelagem de gráficos 2D de grafos.

4.2 Inspector

Os testes iniciais do componente *inspector* aconteceram de forma com que cada vértice e aresta inserida ao painel de modelagem, desse origem a um item na árvore de visualização à direita da tela. Os botões localizados à esquerda do painel de testes, destinariam-se à inserção de vértices e arestas no painel de modelagem e consequentemente ao *inspector*, como observa-se na Figura 4.4, a seguir:



(a) Paineis de modelagem e inspector com apenas um vértice inserido à modelagem.



(b) Após inserção de mais vértices e arestas ao painel de modelagem, pode-se notar a inclusão automática destes no inspector na lateral direita.

Figura 4.4: Teste da implementação do componente inspector.
Fonte:Elaborada pelo autor.

Após validar o funcionamento de tal subcomponente, melhorias em sua parte gráfica

foram realizadas, de forma com que ícones fossem utilizados para indicar seus vértices e arestas.

Por fim, mais testes foram realizados e como forma de representar o grafo modelado em tal componente, prosseguiu-se com o formato de árvore de visualização, ao qual cada item desta árvore representaria um vértice do grafo e seus itens "filhos" indicariam as arestas que se originariam neste vértice, seguidos do nome do vértice de destino. Conforme novos vértices ou arestas seriam inseridos ao grafo, o painel de modelagem inseria novos itens em seu escopo de forma automática. Na figura 4.5 o novo visual do subcomponente e seu funcionamento são representados.

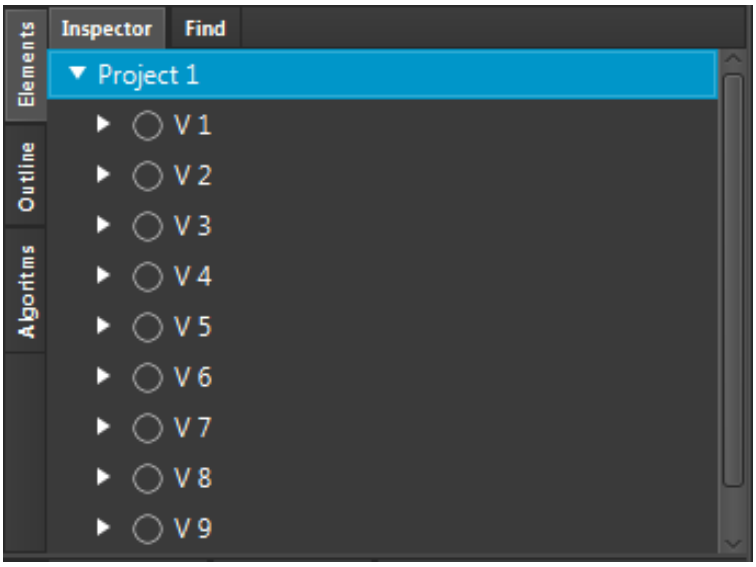


Figura 4.5: VaucangraphFx - Teste do componente Inspector.

Os ícones das arestas a serem utilizados no subcomponente *inspector* variam conforme o tipo de grafo modelado e tipo de aresta utilizada. Os tipos de grafos podem ser direcionados ou não-direcionados. Já as arestas, além das convencionais, podem ser laços ou curvas.

4.3 StatusBar

Este componente tem como objetivo demonstrar as informações inerentes ao painel de modelagem. Na estrutura desse componente, existe a possibilidade de alternância entre o conteúdo mostrado no painel de modelagem e um painel de texto, o que integrará a funcionalidade de transcrição dos grafos modelados para a linguagem do pacote VAUCANSON-G, pertencente ao LaTeX. De tal maneira, almeja-se ampliar o público que utilizará a ferramenta, abrangendo pessoas que necessitem modelar grafos ou autômatos na plataforma LaTeX, mas que possuem pouco ou nenhum conhecimento acerca do pacote VAUCANSON-G. No momento a funcionalidade de transcrição anteriormente citada não foi concluída devido ao término do prazo para desenvolvimento deste trabaho.

A posição do ponteiro do mouse no painel de modelagem seria mostrado e atualizado em tempo real. Informações inerentes ao tamanho do painel de modelagem, selecionados ao

criar o projeto e passível à mudanças durante sua modelagem, também são demonstrados neste componente.

4.4 VaucangraphFx

Embora a aplicação tenha componentes concluídos e funcionais, o prazo para o desenvolvimento da ferramenta, previamente estipulado em um cronograma, esgotou-se e a aplicação encontra-se sem a injeção de dependência necessária para a biblioteca *mvvmFX* interligar os componentes desenvolvidos para que funcionem como um todo. Portanto, a ferramenta tornou-se um arcabouço para que futuras implementações possam adaptá-los conforme sua necessidade e destiná-lo tanto à área de ensino, abrangendo alunos e professores, quanto ao público que faça a utilização da plataforma LaTeX para escrita de documentos científicos e necessite representar grafos ou autômatos utilizando o pacote VauCanSon-G. De tal modo, surge a possibilidade de implementar na ferramenta o necessário para interpretar os grafos ou autômatos modelados e exportá-los em formato de código VauCanSon-G, o que excluiria a necessidade de conhecimento aprofundado relacionado ao pacote citado.

A imagem 4.6 tem como intuito exemplificar o funcionamento dos componentes da VaucangraphFx, embora estes tenham sido testados de forma isolada e unidos na Figura apenas para ilustração.

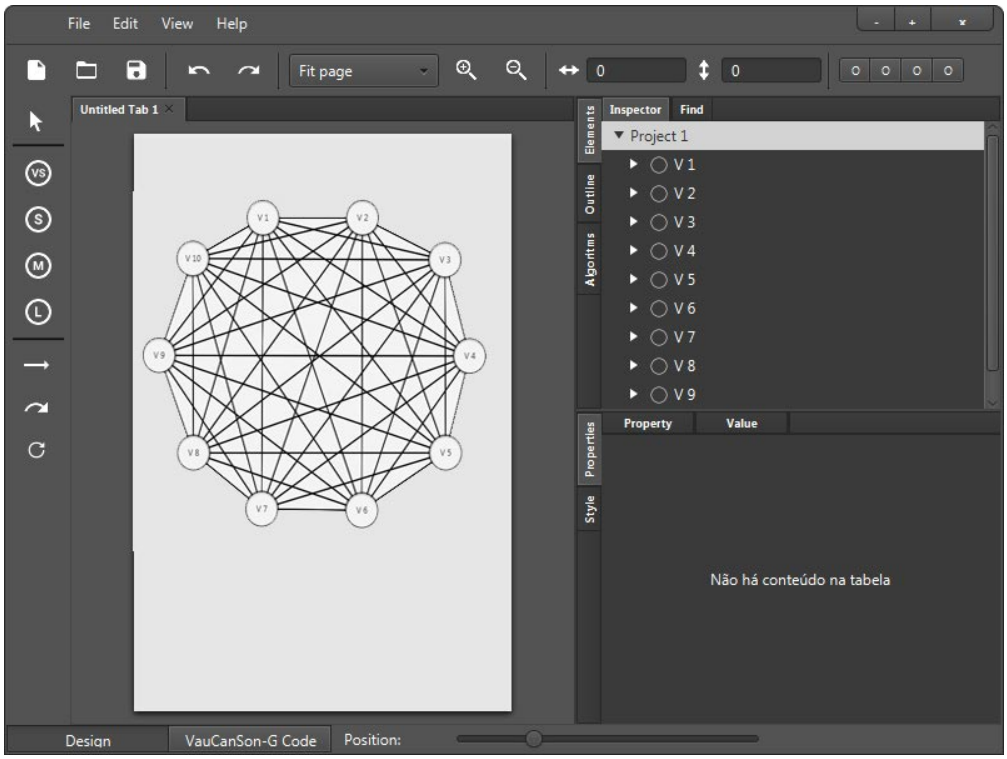


Figura 4.6: Modelagem de um grafo completo K_{10}, utilizando a ferramenta VaucangraphFx.

Assim sendo, as implementações realizadas neste trabalho serão disponibilizadas ao público que tenha interesse em adaptá-lo à sua necessidade, propor alterações ou incluir funcionalidades à aplicação.

Capítulo 5

Considerações Finais

5.1 Conclusão

Atualmente existem alguns softwares para modelagem de grafos e autômatos, entretanto utilizam tecnologias ultrapassadas e interfaces simples, pouco atrativas e que não possuem aparência de uma aplicação profissional. Baseando-se nas informações coletadas do censo de educação superior do ano de 2017, fornecido pelo INEP - Instituto Nacional de Estudos e Pesquisas Educacionais Anísio Teixeira Brasília (2017), 265.396 matrículas foram realizadas somando as áreas de Ciência da computação, Matemática e Processamento de Informação, nas redes privadas e públicas. Considerando que ao menos 1/4 desses alunos matriculados estudem a teoria dos grafos ou autômatos e necessitem de uma ferramenta de apoio, seriam 66.349 pessoas ao ano que poderiam fazer uso da ferramenta proposta neste trabalho, observando que entre os anos de 2007 à 2017 o quantitativo de matrículas teve uma média de crescimento anual de 4,6 % , pressupondo assim, o aumento desses indicadores nos próximos anos.

Sendo assim, neste trabalho foram realizados os estudos de tecnologias necessárias ao desenvolvimento de um software voltado ao auxílio de estudantes e/ou professores da disciplina de Teoria dos Grafos e desenvolvedores que busquem uma ferramenta de apoio para modelagem e manipulação de grafos e autômatos. Para o desenvolvimento da aplicação, foram analisados diversos softwares e suas particularidades, filtrando o necessário para a elaboração de uma ferramenta que atendesse às necessidades do público alvo. De tal forma, iniciaram-se as implementações da ferramenta proposta entretanto, um erro interno ao Java ocasionou o atraso do desenvolvimento deste trabalho e após meses tentando encontrar a solução para tal problema, notou-se que a ferramenta era inexequível com tais tecnologias empregadas, como demonstrado na seção 3.2 deste trabalho.

Por fim, como evidenciado na seção 3.3, a migração das tecnologias adotadas neste projeto foram realizadas. Desta vez a versão modular do Java 11 e a nova versão da biblioteca JavaFx foram utilizadas. Objetivando a redução de dependências no projeto, a remoção da biblioteca JGraphX foi feita e as classes e funcionalidades necessárias à modelagem de grafos e/ou autômatos foram implementadas.

Por utilizar tecnologias recentes e que recebem atualizações periodicamente, o funciona-

mento da aplicação apresenta maior confiabilidade, proporcionando desta maneira, melhor aproveitamento dos recursos oferecidos pelo computador.

5.2 Contribuições

As contribuições deste trabalho são:

- A disponibilização das informações inerentes aos erros que inviabilizaram parte deste trabalho, evidenciados na seção 3.2, evitando que outros desenvolvedores enfrentem os problemas mencionados anteriormente;
- A disponibilização do arcabouço da aplicação, no endereço ⟨https://github.com/Alex-Vieira/VaucangraphFx⟩, que por sua vez oferece a possibilidade de implementação de uma ferramenta destinada ao ensino, uma ferramenta à comunidade de desenvolvedores e pesquisadores da área de grafos;
- A possibilidade de exportação do grafo gerado para códigos VAUCANSON-G, ampliando desta maneira seu público alvo;

5.3 Propostas para trabalhos futuros

Para trabalhos futuros propõe-se:

- Implementação da injeção de dependência entre os componentes desenvolvidos, utilizando a biblioteca *mvvmFX*, atendendo ao padrão de projeto empregado neste trabalho;
- Implementação de algoritmos de análise dos grafos modelados tais como:
  - Algoritmo Dijkstra;
  - Algoritmo de busca em profundidade;
  - Algoritmo de busca em largura;

Bibliografia

ALDER, G. The jgraph tutorial. *Veröffentlicht auf: http://www. jgraph. com/docs. html*, v. 37, p. 62–71, 2001.

BRITCH, D. *Enterprise Application Patterns using Xamarin. Forms*. [S.l.]: Redmond, Washington, DevDiv, .NET and Visual Studio produc teams, 2017.

CASALL, A.; MAUKY, M. *mvvmFX*: Documentation. 2017. Disponível em: ⟨https://github.com/sialcasa/mvvmFX/wiki⟩.

DEL-VECCHIO, R. R. et al. Medidas de centralidade da teoria dos grafos aplicada a fundos de ações no brasil. *XLI Simpósio Brasileiro de Pesquisa Operacional, Porto Seguro, Brasil*, p. 1–4, 2009.

FEOFILOFF, P.; KOHAYAKAWA, Y.; WAKABAYASHI, Y. Uma introdução sucinta à teoria dos grafos. 2011.

GIRALDO, S. R.; ZULUAGA, G. A. O.; ESPINOSA, C. L. Networking en pequeña empresa: una revisión bibliográfica utilizando la teoria de grafos. *FRANCISCO JOSÉ DE CALDAS*, 2014.

INEP - INSTITUTO NACIONAL DE ESTUDOS E PESQUISAS EDUCACIONAIS ANÍSIO TEIXEIRA BRASÍLIA. *Censo da educação superior 2017: sinopse estatística*. 2017. Disponível em: ⟨http://inep.gov.br/censo-da-educacao-superior⟩. Acesso em: 17-06-2019.

INTELLIJ IDEA. *Module dependency diagrams*. 2019. Disponível em: ⟨https://www.jetbrains.com/help/idea/project-module-dependencies-diagram.html\#Project_module_dependencies_diagram.xml⟩. Acesso em: 17-06-2019.

JGRAPH LTD. *JGraphX (JGraph 6) User Manual*. 2019. Disponível em: ⟨https://jgraph.github.io/mxgraph/docs/manual_javavis.html⟩. Acesso em: 17-06-2019.

JURKIEWICZ, S. Grafos–uma introdução. *Programa de Iniciação Cientifica da*, 2009.

LEWIS, H. R.; PAPADIMITRIOU, C. H. *Elementos de Teoria da Computação*. [S.l.: s.n.], 2000.

LOZADA, L. A. P. A-graph: Uma ferramenta computacional de suporte para o ensino-aprendizado da disciplina teoria dos grafos e seus algoritmos. In: *Anais dos Workshops do Congresso Brasileiro de Informática na Educação*. [S.l.: s.n.], 2014. v. 3, n. 1, p. 61.

LUCCHESI, C. L. *Introdução à teoria dos grafos*. [S.l.]: IMPA, 1979.

MENEZES, P. B. *Linguagens Formais e Autômatos: Volume 3 da Série Livros Didáticos Informática UFRGS*. [S.l.]: Bookman Editora, 2009.

MICROSOFT. *O que é o MVVM?* 2019. Disponível em: ⟨https://docs.microsoft.com/pt-br/learn/modules/design-a-mvvm-viewmodel-for-xamarin-forms/2-what-is-mvvm⟩.

MILLER, F. P.; VANDOME, A. F.; MCBREWSTER, J. *Apache Maven*. [S.l.]: Alpha Press, 2010.

PLANK, J. S. Jgraph-a filter for plotting graphs in postscript. In: *USENIX Winter*. [S.l.: s.n.], 1993. p. 61–66.

PRESTES, E. Introdução à teoria dos grafos. *Universidade Federal do Rio Grande do Sul, Instituto de Informática, Departamento de Informática Teórica, Tech. Rep*, 2016.

ROSEN, K. H. *Matemática discreta e suas aplicações*. [S.l.]: Grupo A Educação, 2009.

SANGIORGI, U. B. Rox: Uma ferramenta para o auxílio no aprendizado de teoria dos grafos. *IV ERBASE–Escola Regional de Computação Bahia-Sergipe. Anais do Evento: UEFS*, 2006.

SANTOS, A. P. D. et al. AutÔmatos de estados finitos - revisÃo de literatura. *Revista UniVap*, Universidade do Vale do Paraíba, v. 22, n. 40, February 2017. ISSN 1517-3275. Disponível em: ⟨https://doaj.org/article/aa73a2b8e6d34f69958132420bcd7e1b⟩.

SOUZA, J. A. P. de; KRUGER, K. *Vaucangraph 2.0: A evolução de uma ferramenta gráfica para geração de código Vaucanson-G em documentos LATEX*. Monografia (Trabalho de conclusão de curso) — Sistemas de informação, Universidade Federal de Mato Grosso do sul, Coxim, 2010.

TEIXEIRA, F. *Introdução e boas práticas em UX Design*. [S.l.]: Editora Casa do Código, 2014.

ZIVIANI, N. Projeto de algoritmos com implementação em java e c++. *THOMPSON Learning, São Paulo, 1st edição*, 2007.

ZIVIANI, N. *Projeto de Algoritmos com implementações em java e c++*. [S.l.: s.n.], 2013.