Universidade Federal
do Rio de Janeiro
Escola Politécnica

NETMOSA - UM AMBIENTE PROGRAMÁVEL BASEADO EM LUA PARA SIMULAÇÃO DE MODELOS DE REDE COM ANIMAÇÃO E ESTATÍSTICAS EM TEMPO REAL

Bernardo Dornellas Cysneiros Gomes de Amorim

Projeto de Graduação apresentado ao Curso de Engenharia de Computação e Informação da Escola Politécnica, Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Engenheiro.

Orientador: Daniel Ratton Figueiredo

Rio de Janeiro
Setembro de 2019

Dornellas Cysneiros Gomes de Amorim, Bernardo

Netmosa - Um ambiente programável baseado em Lua para simulação de modelos de rede com animação e estatísticas em tempo real/Bernardo Dornellas Cysneiros Gomes de Amorim. – Rio de Janeiro: UFRJ/ Escola Politécnica, 2019.

XII, 28 p. 29, 7cm.

Orientador: Daniel Ratton Figueiredo

Projeto de Graduação – UFRJ/ Escola Politécnica/ Curso de Engenharia de Computação e Informação, 2019.

Referências Bibliográficas: p. 23 – 24.

1. Animação. 2. Modelos de Rede. 3. Simulação. I. Ratton Figueiredo, Daniel. II. Universidade Federal do Rio de Janeiro, Escola Politécnica, Curso de Engenharia de Computação e Informação. III. Título.

NETMOSA - UM AMBIENTE PROGRAMÁVEL BASEADO EM LUA PARA SIMULAÇÃO DE MODELOS DE REDE COM ANIMAÇÃO E ESTATÍSTICAS EM TEMPO REAL

Bernardo Dornellas Cysneiros Gomes de Amorim

PROJETO DE GRADUAÇÃO SUBMETIDO AO CORPO DOCENTE DO CURSO DE ENGENHARIA DE COMPUTAÇÃO E INFORMAÇÃO DA ESCOLA POLITÉCNICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO DE COMPUTAÇÃO.

Examinado por:

Prof. Daniel Ratton Figueiredo, PhD

Prof. Fabio Happ Botler, DSc.

Prof. Priscila Machado Vieira Lima, PhD

RIO DE JANEIRO, RJ – BRASIL
SETEMBRO DE 2019

Dedico este trabalho a minha atual noiva e futura esposa, Anne Leite, por todo apoio e incentivo dado ao longo destes últimos 6 anos.

Agradecimentos

Agradeço aqui às pessoas que fizeram parte do meu trajeto acadêmico e deste trabalho, direta ou indiretamente:

Ao meu orientador, Daniel Ratton Figueiredo, por não apenas orientação acadêmica mas pela paciência, compreensão, apoio e conselho para até assuntos além da universidade.

À minha noiva, Anne Caroline Reis Leite, por me fazer acreditar em mim mesmo até nos momentos mais difíceis e estar sempre do meu lado.

Aos meus pais, Filipa de Dornellas Cysneiros e Francisco de Almeida Gomes de Amorim, por toda base que tive antes de ingressar na universidade, pelo apoio nos momentos difíceis, por nunca duvidar de mim e pela formação do meu caráter.

Ao Igor Macedo Quintanilha pelo companheirismo, por me incentivar e me inspirar no âmbito acadêmico e também por sempre ter as conversas difíceis, falando o que eu precisava ouvir e não o que eu queria.

Ao Pedro Volpi Nacif por tornar as vindas a faculdade mais divertidas, por não abaixar a cabeça frente a abusos de autoridade dentro da universidade e por me ajudar a ver que eu era bom no que fazia.

Ao Sibelius Claussen Dantas por tornar mais divertidos todos os anos que moramos juntos e por servir de assistente psicológico de plantão nos momentos difíceis.

Ao Rodrigo Lima Veloso por sempre mostrar o quão importante e valiosas são as amizades.

Ao Rafael Rodrigues por usar sua mágica para tornar o Netmosa mais bonito e fácil de usar.

Resumo do Projeto de Graduação apresentado à Escola Politécnica/ UFRJ como parte dos requisitos necessários para a obtenção do grau de Engenheiro de Computação.

NETMOSA - UM AMBIENTE PROGRAMÁVEL BASEADO EM LUA PARA SIMULAÇÃO DE MODELOS DE REDE COM ANIMAÇÃO E ESTATÍSTICAS EM TEMPO REAL

Bernardo Dornellas Cysneiros Gomes de Amorim

Setembro/2019

Orientador: Daniel Ratton Figueiredo

Curso: Engenharia de Computação e Informação

Netmosa é uma ferramenta que permite descrever um modelo de rede utilizando a linguagem Lua e com isso visualizar a rede evoluindo em tempo real. O foco da ferramenta é ajudar a entender modelos de rede de maneira visual e intuitiva, através dois modos de visualização: a própria estrutura da rede, organizada através de um *force-model*; e a visualização de algumas distribuições, como o grau dos vértices. Por fim, a ferramenta permite exportar, para análises posteriores, os dados das distribuições em CSV ou a rede em si em GraphML.

Palavras-chave: animação, modelos de rede, simulação.

Abstract of Undergraduate Project presented to POLI/UFRJ as a partial fulfillment of the requirements for the degree of Engineer.

NETMOSA - A PROGRAMMABLE ENVIRONMENT BASED ON LUA TO SIMULATE NETWORK MODELS WITH ANIMATION AND REAL TIME STATISTICS

Bernardo Dornellas Cysneiros Gomes de Amorim

September/2019

Advisor: Daniel Ratton Figueiredo

Course: Computer Engineering

Netmosa is a tool that allow users to describe a network model in Lua and then visualize in real time the evolving network. The focus of the tool is to ease understanding of network models in a visual and intuitive way. It offers two views of the network: it's own structure, organized with a force-model simulation; and some network and node distributions, such as the degree distribution. On top of that, it allows the user to export the state of the network at any time in GraphML or the distributions in CSV.

Keywords: animation, network model, simulation.

Sumário

Lista de Figuras

Lista de Algoritmos

Lista de Abreviações

HTML	HyperText Markup Language
CSS	Cascading Style Sheets
CSV	Comma-Separated Values
PDF	Probability Distribution Function
CDF	Cumulative Distribution Function
CCDF	Complementary Cumulative Distribution Function

Capítulo 1

Introdução

1.1 Tema

O tema deste trabalho foi desenvolver uma aplicação web que tivesse a capacidade de auxiliar em uma melhor compreensão dos modelos de rede através da especificação, simulação e visualização animada destes modelos em tempo real.

1.2 Delimitação

Durante o desenvolvimento do trabalho, pontos possíveis de expansão do escopo foram pensados e descartados. O projeto tem como foco conseguir simular grafos não direcionais e com centenas de vértices, apesar de não existir um limite do tamanho da rede. O foco era na prova do conceito para que pudesse ser melhorado e expandido em trabalhos futuros.

1.3 Justificativa

O estudo e desenvolvimento de modelos de rede é um tema quente no mundo atual. Esses modelos podem ajudar a entender como as relações entre pessoas, proteínas ou doenças emergem e não apenas como elas se parecem.

Este assunto já foi tema de disciplinas na universidade, como a de Redes Complexas, e e já foi abordado em diversos congressos. Alguns artigos sobre o assunto já foram citadas dezenas de milhares de vezes, como o BARABÁSI e ALBERT [1] e ERDŐS e RÉNYI [2].

Uma ferramenta focada no estudo e entendimento de modelos de redes pode ajudar esta área do conhecimento a evoluir ainda mais, desde o ensino até a pesquisa.

1.4 Objetivos

O objetivo deste trabalho é disponibilizar para pesquisadores, professores e alunos interessados em modelos de rede uma ferramenta que possa fazer parte do processo de pesquisa, ensino ou aprendizado. Uma ferramenta agnóstica a sistemas operacionais e sem a necessidade de instalação de softwares extras.

Essa ferramenta deve permitir a especificação do modelo de rede através de uma linguagem de programação, para que essa especificação formal possa ser executada e visualizada em tempo real com animações e estatísticas sobre a rede. Com isso, esses dados dão ao usuário uma intuição sobre como o modelo se comporta e podem explicar fenômenos resultantes.

1.5 Metodologia

O desenvolvimento deste trabalho foi feito em duas partes.

A primeira foi o desenvolvimento e validação da aplicação. Anteriormente a este trabalho, uma ferramenta similar, porém com escopo reduzido, foi desenvolvida durante um trabalho de iniciação científica. Para este trabalho, o objetivo era expandir o que havia sido feito e criar uma ferramenta que pudesse ser usada por uma audiência maior. Logo, o escopo do projeto era bem definido e a dificuldade maior eram os desafios tecnológicos e de interface hómem-máquina. Portanto, o processo foi pouco estruturado e mais experimental, com o objetivo de explorar as possíveis formas de se desenvolver a aplicação desejada.

A segunda etapa foi o momento de escrever este documento e organizar as ideias. Como a primeira etapa foi feita de maneira experimental, era preciso organizar o projeto para poder detalhar melhor o seu funcionamento. Tendo isto em mente, o código foi reorganizado e arquitetura do sistema foi repensada durante o processo de escrita deste documento. O ato de o descrever ajudou a entendê-lo melhor.

1.6 Organização do Trabalho

No Capítulo 2, é introduzido a informação necessária para o entendimento do projeto e do seu desenvolvimento. No Capítulo 3, são introduzidas ferramentas similares ou relacionadas à aplicação deste projeto. No Capítulo 4, é apresentada a ferramenta desenvolvida neste projeto e comparada com as demais existentes, demonstrando o funcionamento e a interface. No Capítulo 5, é descrito, em mais detalhes, a parte técnica da aplicação e como ela foi desenvolvida. Por último, no Capítulo 6, é feita uma análise do resultado obtido e possíveis trabalhos futuros.

Capítulo 2

Fundamentação Teórica

2.1 Modelos de redes aleatórias

O mundo é complexo e difícil de entender. Há tempos se criam modelos simplificados para facilitar o entendimento do que está ao nosso redor. Redes, ou grafos, são um modelo matemático simples, porém com uma capacidade de representar sistemas complexos.

Redes então, são um conjunto de objetos, chamados de vértices ou nós, conectados por arestas ou arcos. Entretanto, existem diversas variações desse modelo.

Em alguns as arestas, então chamadas de arcos, tem direção. Isto é, um arco entre os vértices v_1 e v_2 é diferente do arco entre o v_2 e v_1. Em alguns modelos, vértices e arestas podem conter mais informações, como por exemplo, um peso numa aresta para diferenciar conexões mais importantes.

Uma das áreas de estudo de redes é a área de modelos de redes aleatórias. Esta área tenta descrever processos para gerar grafos aleatórios de maneira que algumas propriedades ou métricas sejam observadas.

Alguns modelos foram implementados neste trabalho como exemplos e por isto serão introduzidos aqui também.

2.1.1 Modelos Erdös e Rényi

Entre os modelos mais estudados estão os modelos chamados de Erdös e Rényi, que referem normalmente a uma de duas variações: o $G(n, M)$[3], que foi introduzido por ERDÖS e RÉNYI; e o $G(n, P)$[4], introduzido no mesmo ano, de maneira independente, por GILBERT.

O modelo mais comum, o $G(n, P)$ consiste em gerar um grafo com n vértices onde todas as $n(n-1)/2$ possíveis arestas existem com probabilidade p. Um pseudo-código para o modelo pode ser visto no Algoritmo 2.1.

Entrada: n, P
Saída: Um grafo com n nós é gerado
início
  $V = []$
  $E = []$
  para *cada* $v \in [1, n]$ **faça**
    $V_v =$ um novo vértice
  fim
  para *cada* $i \in [1, n]$ **faça**
    para *cada* $j \in [i + 1, n]$ **faça**
      $k =$ número aletório uniforme entre 0 e 1
      se $k < P$ **então**
        $E_{|E|} = (i, j)$
      fim
    fim
  fim
fim

Algoritmo 2.1: $G(n, P)$

2.1.2 Modelos de crescimento de rede

Muitos dos modelos tradicionais mais estudados consistiam em gerar grafos com número determinado de vértices. Isto é um problema, pois não explica como a rede pode ter surgido ao longo do tempo. Modelos de crescimento de rede, portanto, não precisam de um número de vértices pré-determinado, pois podem ser executados indefinidamente, com o tamanho da rede crescendo com o tempo.

Modelo Barabàsi-Albert

Em 1999, BARABÁSI e ALBERT descreveram um modelo de crescimento de rede baseado na ideia de anexação preferencial[1].

O modelo começa com um clique com m_0 vértices e a cada etapa um novo vértice é ser adicionado a rede e é conectado a m vértices já existentes com probabilidade proporcional ao grau do vértice.

O que acontece então é que quanto maior o grau de um vértice, maior a probabilidade do grau dele aumentar. Isto é o fenômeno de anexação preferencial.

Por ser um modelo de crescimento, pode-se entender que gerar um grafo de n vértices é na verdade a continuação de um processo onde se gerou um grafo de m vértices onde $n > m$.

Como pode ser difícil obter aleatoriamente um vértice proporcional ao seu grau, um método alternativo é selecionar uma aresta de maneira uniformemente aleatória e depois escolher uma das duas pontas de aresta com probabilidade igual. Isso é equivalente à escolher o vértice proporcionalmente

O algoritmo do modelo em pseudo código pode ser visto no algoritmo 2.2

Entrada: m_0, m, n
Saída: Um grafo com n nós é gerado
início
  $V = []$
  $E = []$
  para *cada* $v \in [1, m_0]$ **faça**
    $V_v =$ um novo vértice
    para *cada* $u \in [1, v-1]$ **faça**
      $E_{|E|} = (u, v)$
    fim
  fim
  para *cada* $v \in [m_0 + 1, n]$ **faça**
    para *cada* $i \in [1, m]$ **faça**
      $e =$ aresta escolhida de maneira uniforme em E
      $k =$ número aletório uniforme entre 0 e 1
      se $k > 0.5$ **então**
        $E_{|E|} = (v, e_1)$
      senão
        $E_{|E|} = (v, e_2)$
      fim
    fim
  fim
fim

Algoritmo 2.2: Barabàsi-Albert

Modelos baseados em passeios aleatórios

Dois modelos dados como exemplo na ferramenta são baseados em passeios aleatórios: o "No Restarting Random Walk"[5], descrito em 2.3; e o "Bernouli Growth Random Walk"[6], descrito em 2.4.

2.2 Métricas sobre redes

O estudo de redes e seus modelos tenta entender, entre outras coisas, a estrutura da rede gerada e suas propriedades. Entretanto, não basta olhar a figura de uma rede gerada através de um modelo e assumir que entendemos a estrutura de todas as redes geradas pelo mesmo, visto que trata-se de um modelo estocástico no qual cada rede gerada será diferente.

Do mesmo modo que observar o valor obtido ao rolar um dado de seis lados não dirá nada sobre o processo de rolar dados. O que precisamos é uma forma de entender, em linhas gerais, o que o processo nos dá. Por exemplo, entender que a média do valor dos dados será 3.5.

No caso de estudo de redes, normalmente queremos observar algumas métricas sobre a rede, sobre os vértices da rede ou sobre conjuntos de vértices. Como por

Entrada: k, n
Saída: Um grafo com n nós é gerado
início
  $V = []$
  $E = []$
  $W = 1$
  $V_1 =$ um novo vértice
  $E_{|E|} = (1, 1)$
  para *cada* $v \in [1, n]$ **faça**
    para *cada* $i \in [1, k]$ **faça**
      $W =$ vizinho de W selecionado uniformemente
    fim
    $V_v =$ um novo vértice
    $E_{|E|} = (v, W)$
  fim
fim

Algoritmo 2.3: No Restarting Random Walk

Entrada: p, n
Saída: Um grafo com n nós é gerado
início
  $V = []$
  $E = []$
  $W = 1$
  $v = 1$
  $V_v =$ um novo vértice
  $E_{|E|} = (1, 1)$
  repita
    $W =$ vizinho de W selecionado uniformemente
    $k =$ número aletório uniforme entre 0 e 1
    se $k < p$ **então**
      $v = v + 1$
      $V_v =$ um novo vértice
      $E_{|E|} = (v, W)$
    fim
  até $v = n$;
fim

Algoritmo 2.4: Bernouli Growth Random Walk

exemplo, distribuição de grau, distância média dos vértices ou probilidade de gerar cliques de tamanho N.

Neste trabalho, dois coletores de métricas foram desenvolvidos para a ferramenta: distribuição de grau e distribuição da distância do primeiro vértice.

2.3 Visualização de redes

Por mais que métricas deem informações descritivas sobre as redes geradas, isto pode não ser o suficiente para entender o modelo e talvez correlacionar com fenômenos naturais. Para isso, uma parte da pesquisa de redes estuda a visualização das mesmas.

A visualização trata de transformar um grafo numa imagem em duas dimensões com o objetivo de dar insumos intuitivos para aumentar a compreensão sobre a rede ou modelo de rede.

Existem diversas técnicas para realizartal tarefa. Uma das classes mais utilizadas é a visualização baseada simulação física de forças entre corpos. Esta técnica consiste em uma simulação física na qual vértices interagem um com os outros com forças repulsivas e atrativas, sendo que vértices conectados se atraem enquanto não conectados se repelem.

2.4 Aplicações Web

Aplicações Web, embora não especificadas formalmente, referem a aplicações que são armazenadas em um servidor enviadas à um programa chamado de navegador através do protocolo `HTTP`.

O navegador então se encarrega de exibir a aplicação para o usuário sem que ele tenha que obter nem instalar nenhuma outra aplicação ou programa de maneria explícita. De certa maneira, a aplicação é obtida durante o acesso.

2.5 Programação Reativa e RxJS

Programação reativa é um paradigma de programação baseado em fluxos de dados. A ideia central é que os componentes de um programa se comuniquem principalmente através de dados gravados nestes fluxos num modelo produtor consumidor. O que muda é que, se um componente A depende de um estado num componente B, o modelo tradicional requer que ou B saiba da existência de A para comunicá-lo sobre a mudança ou que A fique verificando o estado de B com frequência para se atualizar. No modelo reativo, A se subscreve no fluxo de dados de B contendo o estado que ele está interessado e receberá automaticamente as notificações de mudança de estado.

A diferença principal da ideia de programação reativa para um simples produtor consumidor, é que estes fluxos de dados podem persistir dados anteriores e são objetos principais no sistema, podendo ser combinados e modificados com algumas funções, de maneira declarativa.

Neste projeto utilizamos a biblitoeca `RxJS`[7] para implementar programação reativa.

2.6 ReactJS

`ReactJS`[8], como definida na sua página principal, é "uma biblioteca JavaScript para criar interfaces de usuário", e a definem como declarativa e baseada em componentes. Ela é baseada em componentes pois cada parte da interface é um componente (ou um grupo de componentes), onde cada componente carrega consigo um estado e valores que foram passados de componente pai para componente filho. Com os dados de estado e as informações extras que lhe foram passadas, é o suficiente para o componente saber se desenhar na tela.

Ao trabalhar com interfaces normalmente é preciso muita preocupação com o ciclo de vida dos objetos utilizados para desenhar a interface. É preciso pensar quando é possível desenhar um componente na tela, quando é que podemos desalocar o espaço de memória e destruir o componente, etc.

`React` é declarativo pois este ciclo de vida é controlado pela biblioteca e o usuário apenas tem que dizer quais componentes são utilizado em cada outro componente. Embora seja possível controlar o que cada componente faz em cada parte do ciclo de vida, grande parte das vezes não é necessário, fazendo com que `React` seja uma ferramenta muito simples para criar interfaces.

2.7 D3.js

`D3.js`[9] é uma biblioteca de visualização de dados declarativa. Ela ajuda a combinar dados com documentos web para criar interfaces interativas. Além disso, esta biblioteca tem diversas ferramentas prontas para visualização, como um modelo físico de forças para organizar nós na tela, facilitando a visualização de grafos.

Capítulo 3

Ferramentas de análise e visualização de redes

Netmosa não é e nem será a única ferramenta para visualizar redes. Entretanto, é uma das primeiras com foco em visualização e animação em tempo real.

De qualquer maneira, a ferramenta foi inspirada em algumas ferramentas já existentes descritas nas seções abaixo.

3.1 Pacotes Python

`NetworkX`[10] e `Graph-tool`[11] são dois dos mais famosos pacotes `Python` para análise de redes complexas.

Os pacotes abrangem muito mais do que visualização, mas também servem para gerar visualizações com diversos algoritmos de *layout*.

3.2 Aplicações Independentes

`Gephi`[12], `Pajek`[13] e `Cytoscape`[14] são aplicações independentes de análise e visualização de redes disponíveis para `Windows` e, no caso do `Gephi` e `Cytoscape`, também `MacOS` e `Linux`.

Estas, diferente dos pacotes `Python` citados acima, são programas muito mais completos para todo o processo de análise de redes. `Cytoscape` inclusive conta com uma série de `Apps` para ajudar com alguns domínios específicos de redes complexas.

Além disso, estes programas contam com visualização em tempo real, podendo mover, selecionar e inspecionar a rede, dando maior flexibilidade para explorar uma rede com muitos vértices.

3.3 Outros

`GraphViz`[15] é uma ferramenta para gerar imagens em duas dimensões de grafos a partir de uma entrada no formato `DOT`.

Diferente das ferramentas acima, `Graphviz` é apenas uma ferramenta para gerar as imagens, sem toda a parte de análise por trás. Além disso, os algorítmos de *layout* são bem limitados, tornando difícil visualização de grandes redes.

Capítulo 4

Netmosa - Network Model Specification and Animation

4.1 Visão geral

Netmosa é uma aplicação web, isto é, utilizada através de um navegador e disponibilizada online através da página `https://bamorim.github.io/netmosa/`.

A aplicação é uma ferramenta para ajudar pesquisadores na área de redes complexas a obterem intuição sobre modelos de geração de redes aleatórias. Para ser uma ferramenta genérica que possa funcionar com qualquer modelo, o modelo pode ser programado usando a linguagem Lua. Com o modelo pronto, é possível iniciar a simulação e acompanhar duas visualizações diferentes: um modelo de força do grafo e gráficos da distribuição da distância para o primeiro vértice e distribuição de grau entre os vértices.

Por possibilitar a visualização em tempo real, a ferramenta ajuda com a intuição de como os modelos funcionam, pois em vez de esperar cada rodada de simulação terminar, o usuário pode acompanhar a evolução da rede passo a passo.

Como um extra, é possível exportar as distribuições em formato `CSV` e o grafo em formato GraphML.

4.2 Editor de código

O editor de código é a tela inicial da aplicação. É nesta tela que o usuário pode programar seu modelo. O editor de código é o `Monaco Editor`, o mesmo por trás do editor amplamente utilizado, o `Visual Studio Code`.

Nesta tela também é possível carregar exemplos de modelos já programados na aplicação e acessar a documentação de programação de modelos.

E por fim, também é possível iniciar a simulação, o que leva para a página de

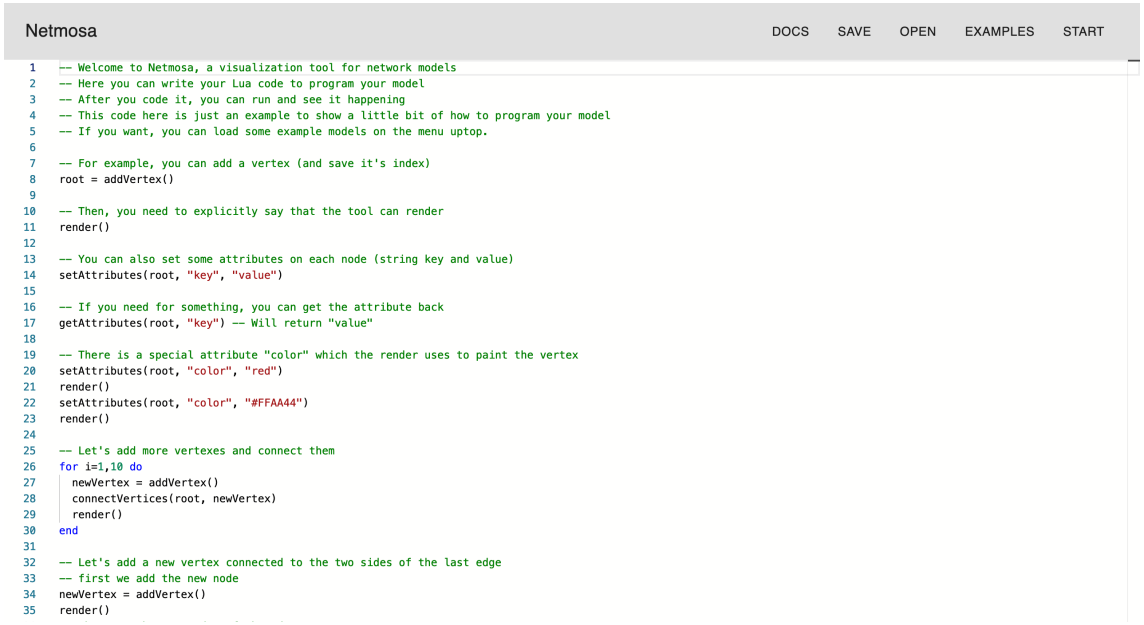

Figura 4.1: Captura de tela: Editor de código

visualização da rede.

4.3 Linguagem Lua

A linguagem Lua foi escolhida como linguagem de programação para os modelos por ter uma sintaxe simples e ser facilmente integrada a outras linguagens. Como descrito pelo paper LERUSALIMSCHY *et al.* [16], `Lua` foi feito para ser uma linguagem para extender aplicações, que é exatamente o caso de uso aqui.

Alguns detalhes da linguagem podem ser estranhos para programadores assíduos, mas foram pensadas para pessoas com menos experiência em programação. Um exemplo é que os vetores são indexados a partir de 1 e não de 0, como em outras linguagens mais populares.

Além disso, a linguagem conta com uma biblioteca padrão bem completa, contando com um módulo de matemática com diversas funções que podem ser úteis na programação de modelos específicos, diminuindo a quantidade de código necessário.

4.4 Visualização da rede

Ao iniciar a simulação, a primeira tela exibida é a de visualização da rede. Nesta tela, é possível ver a estrutura da rede, vértice por vértice. Além disso, a ferramenta também utiliza um atributo especial do vértice, o `color`, para pintar o vértice. Isso é útil para destacar propriedades de forma simples, como por exemplo, a posição de um passeio aleatório.

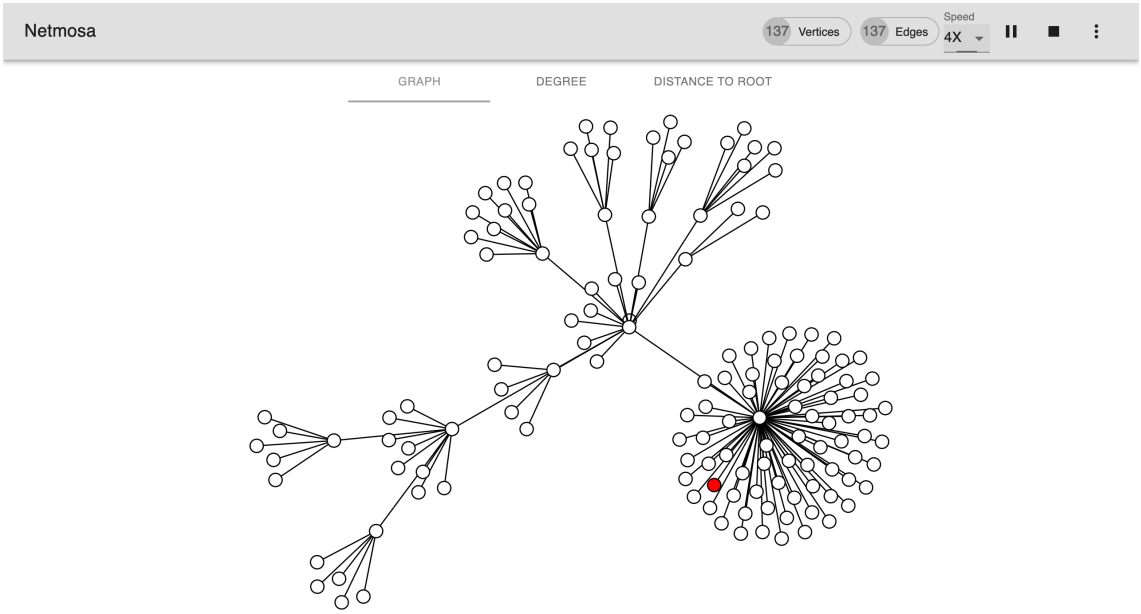

Figura 4.2: Captura de tela: Visualização da rede

4.5 Visualização das métricas

Como alternativa, o usuário pode alterar entre outras duas visualizações de métricas. Estas são a de distribuição de grau e distribuição da distância para o primeiro vértice.

Estas duas telas são identicas, permitindo o usuário configurar a escala dos eixos e as transformações (`PDF`, `CDF` ou `CCDF`) e informando o mínimo, máximo e média.

Para permitir o usuário fazer análises posteriores a simulação, esta tela também fornece a exportação da distribuição em formato `CSV`.

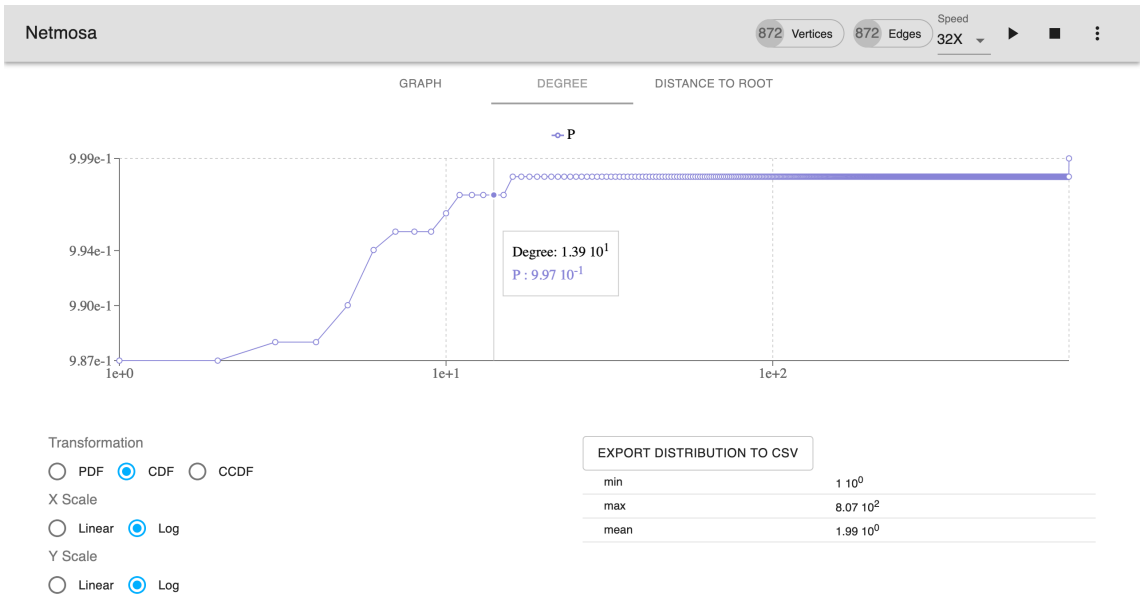

Figura 4.3: Captura de tela: Visualização da distribuição de grau

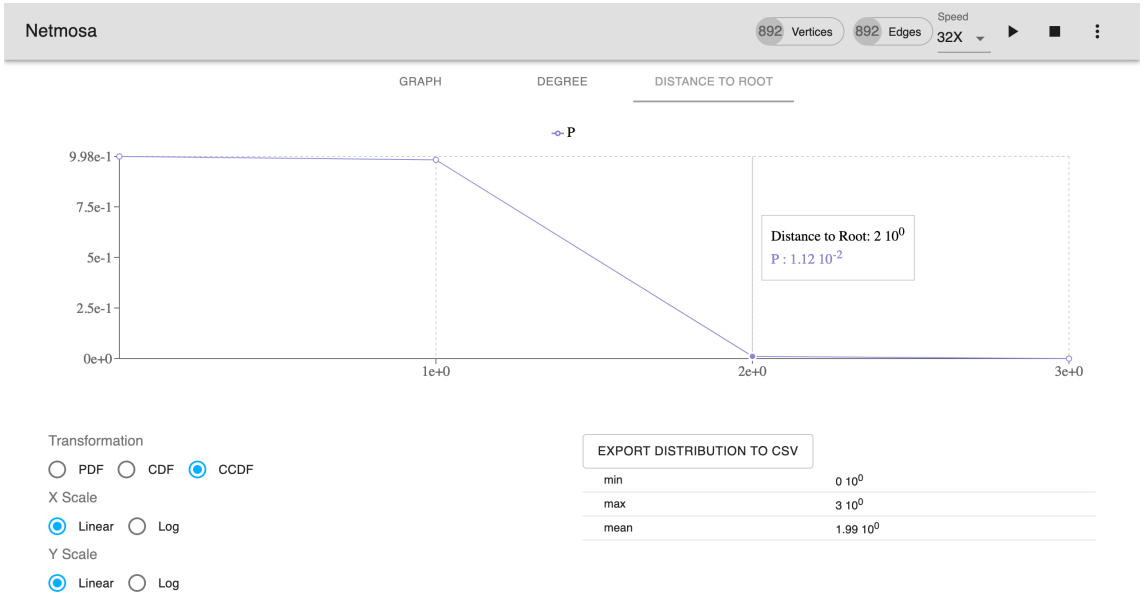

Figura 4.4: Captura de tela: Visualização da distribuição da distância pro primeiro vértice

Capítulo 5

Implementação

Netmosa é uma aplicação web, isto é, executada através de um navegador. Sendo assim, a escolha de tecnologia é relativamente limitada por enquanto. Isso significa que as aplicações devem ter uma interface feita utilizando `HTML` e `CSS` e que utiliza `JavaScript` para as partes interativas.

Entretanto, muitas linguagens, para poderem aproveitar a plataforma web, possuem compiladores que tem como saída código em `JavaScript`, logo, é possível escrever código em outra linguagem e ter `JavaScript` como produto final.

Para este trabalho foi escolhido `TypeScript`[17], que é uma extensão da linguagem `JavaScript` com algumas funcionalidades próprias e com checagem estática de tipos, que permite capturar muitos defeitos em tempo de compilação.

Desde o princípio, a ideia era ter a possibilidade de programar os modelos usando uma linguagem de fácil uso e apenas inserir no runtime dela uma biblioteca de funções para manipular o grafo. Para isso, foi escolhido a linguagem `Lua`[16], por ser uma linguagem feita para ser integrada a outras e por existir uma implementação de uma máquina virtual `Lua` em `JavaScript` chamada `Fengari`[18].

5.1 Arquitetura geral

Netmosa é separado em 2 grandes partes: `Modelo` e `Interface`.

O `Modelo` contém as classes que modelam o estado principal da aplicação, desde o texto escrito no editor até o estado da simulação (bem como o código do simulador em si).

A `Interface`, como o nome diz, é responsável pela interface do usuário e suas interações. A interface captura as interações do usuário, entende os intuitos e envia comandos para o `Modelo` e recebe, como consumidor dos fluxos de dados do `Modelo`, as modificações no estado da aplicação para finalmente exibir a interface atualizada. As tecnologias principais usadas aqui são `ReactJS`, `D3.js` e `Monaco Editor`[19] como editor de código.

5.2 Modelo

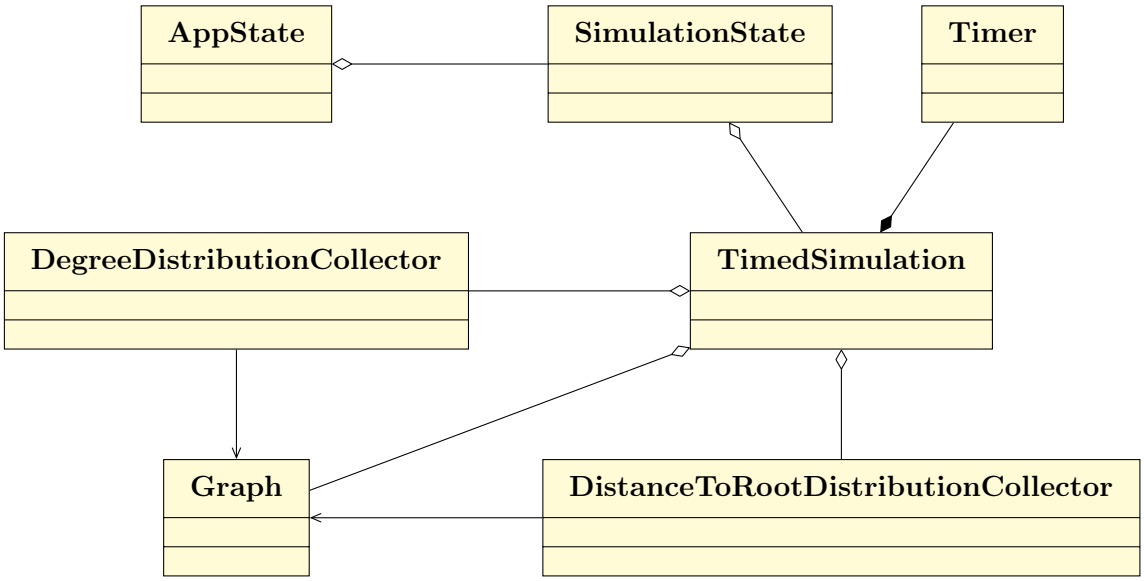

Figura 5.1: Diagramas de classe do `Modelo`

O `Modelo` é composto por classes simples e com pouca interação com bibliotecas externas (apenas com `Fengari`, para interpretar código `Lua`, e RxJS, para os fluxos de dados reativos).

5.2.1 AppState

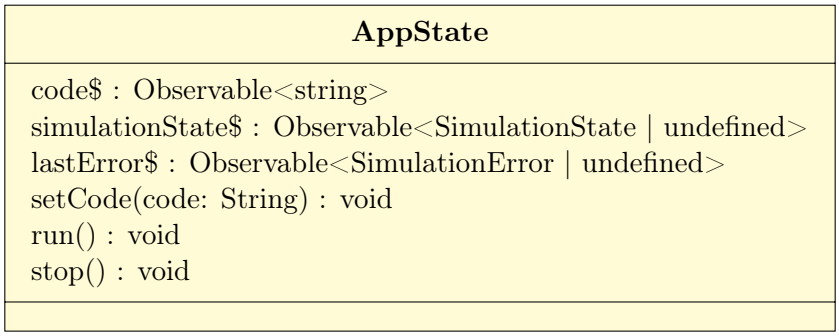

Figura 5.2: Detalhe da classe AppState

`AppState` é a classe principal do `Modelo`. Ela carrega o estado de mais alto nível da aplicação e permite iniciar ou parar as simulações. Quando o método `run` é chamado, uma nova instância do estado da simulação (`SimulationState`) é criada e publicada para os interessados. Quando o método `stop` é chamado, a simulação atual é destruída, e é publicado que não existe nenhuma simulação no momento para os interessados.

Além disso, ela observa os erros publicados pela a simulação e quando estes acontecem, a simulação é terminada da mesma maneira que o método `stop`, com

a diferença de que é publicado também que houve um erro na simulação. Esta informação tem a finalidade de indicar para a interface o erro para que esta possa exibi-lo no local apropriado.

5.2.2 SimulationState

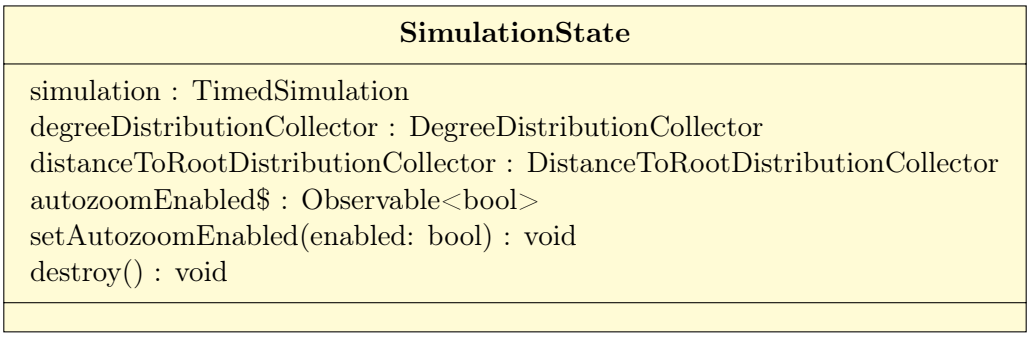

Figura 5.3: Detalhe da classe SimulationState

`SimulationState` é a classe instanciada apenas quando a aplicação possui uma simulação sendo executada. Portanto, contém a própria simulaçào (TimedSimulation) e os coletores de métricas (`DegreeDistributionCollector` e `DistanceToRootDistributionCollector`). Além disso, carrega também um estado compartilhado que é a configuração de autozoom. Esta funcionalidade faz com que conforme a rede cresce a visualização fique mais distante com o objetivo de fazer todos os nós aparecerem na tela, mas que pode ser desabilitada para uma visualizaçào numa parte mais específica. Ela pode ser desligada para facilitar a exploração da rede mas sempre é iniciada habilitada numa simulação, por isso, isto contida na classe `SimulationState` e não na classe `AppState`.

5.2.3 TimedSimulation

TimedSimulation é a classe central do sistema. É ela que interpreta o código `Lua` de maneira temporizada. Para isso, ela cria um objeto da classe Timer (que funciona chamando um método de tempos em tempos, com frequência controlável).

Esta classe, ao instânciar um objeto, inicia também um objeto de grafo (classe Graph) e dá início a simulação, passando o grafo como argumento e fazendo com que cada chamada do Timer chame o próximo passo da simulação.

A simulação é implementada envelopando o código `Lua` do usuário em uma corotina `Lua`. Isto é enviado ao `Fengari`, junto com uma biblioteca padrão. Esta biblioteca padrão consiste principalmente dos métodos sobre o grafo (classe `Graph`) e uma função especial chamada `render`, que é apenas um outro nome para `coroutine.yield` do

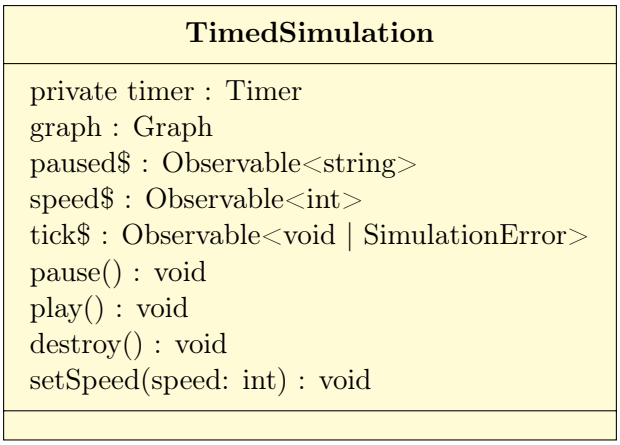

Figura 5.4: Detalhe da classe TimedSimulation

`Lua`. Isto porque, o simulador ao receber o controle novamente irá emitir um novo `tick` para instruir a interface a redesenhar o grafo.

Esta forma de integração configura um modelo multi-tarefa cooperativo, que tem como problema principal o fato de que um usuário pode facilmente escrever um código que trava o simulador num laço infinito. Resolver esse problema fica como trabalho futuro.

Como últimas responsabilidades desta classe, ela também permite parar temporariamente e alterar a velocidade da simulação e publica estas mudanças para a interface conseguir sincronizar. Por último, publica também, um fluxo de *ticks*, isto é, para notificar que a simulação executou mais uma vez, para ajudar na sincronização da interface.

5.2.4 Graph

`Graph` é a classe que modela, como o nome diz, o grafo. A estrutura de dados utilizada é o vetor de lista de adjacência, junto com um vetor de arestas para facilitar a seleção aleatória uniforme de arestas. Os vértices são identificados pelo índice no vetor e podem possuir, além das arestas, atributos. Estes atributos são conjuntos de chave e valor, que servem para o usuário adicionar informações extras aos nós (como por exemplo, cor).

Além disso, é definido também o conceito de `GraphEvent`, que pode ser vértice adicionado, aresta adicionada ou atributo modificado. A cada chamada de métodos, um destes eventos é publicado para as partes interessadas. Isso garante a sincronização do estado do grafo com a interface do usuário e outros componentes importantes.

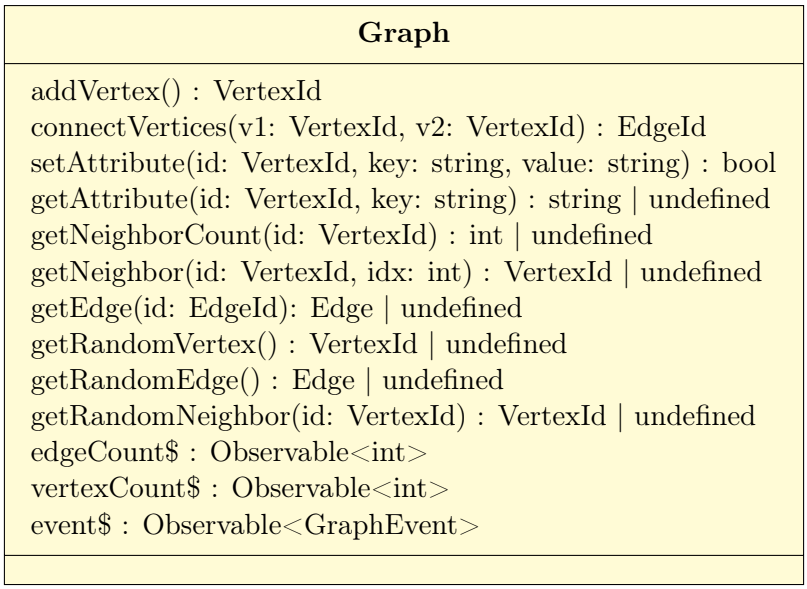

Figura 5.5: Detalhe da classe Graph

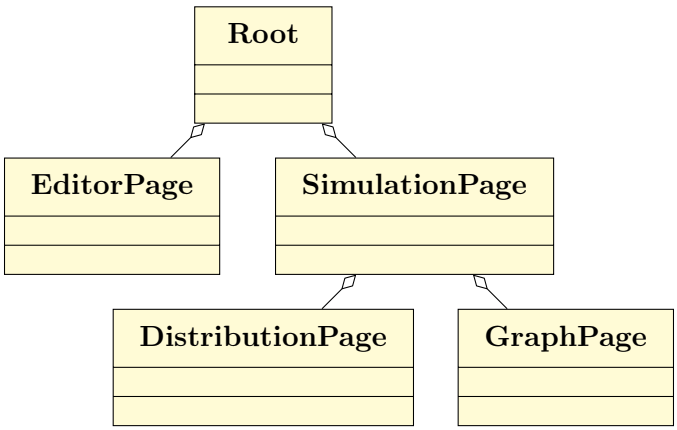

Figura 5.6: Componentes Principais da Interface

5.3 Interface

A `Interface`, como dito acima, é uma aplicação `ReactJS` composta de uma série de componentes para representar cada parte da `Interface` do usuário. É feito de diversos componentes, desde um componente simples para exibir números em notação científica até o componente que desenha o grafo na tela.

Os componentes principais decidem o que exibir baseado no estado da aplicação dado pelas classes do `Modelo`.

5.3.1 Root

O componente `Root` observa as mudanças do `SimulationState` no objeto `AppState` e, quando existe um `SimulationState` instânciado, ele exibe o componente `SimulationPage` e quando não existe, exibe o componente `EditorPage`.

5.3.2 EditorPage

O componente `EditorPage` é responsável por exibir a tela inicial da aplicação, que é o editor de código. Ele utiliza a biblioteca `Monaco Editor` para poder criar um editor de código na aplicação e sincroniza o código com o estado global da aplicação. Além disso, captura outras interações como a de iniciar a simulação.

5.3.3 SimulationPage

O componente `SimulationPage` é responsável por exibir as informações da simulação enquanto executa. Ele exibe os controles globais (como o de velocidade, parar temporariamente e terminar a simulação) e utiliza os componentes `DistributionPage` e `GraphPage` para exibir as duas principais formas de visualizar a simulação.

5.3.4 GraphPage

Este componente é um dos mais importantes da aplicação, pois é exatamente este que permite a visualização interativa de um grafo. Este componente utiliza `D3.js` para criar uma visualização de grafo baseada em um modelo de força.

Para sincronizar a visualização com a simulação, este componente observa os `GraphEvent` publicados pelo `Graph` do `Modelo` e atualiza seu estado interno. Para garantir sincronização na exibição, ele agrega o fluxo de `GraphEvent` utilizando o fluxo de *ticks*, oriundo do objeto `TimedSimulation`. Assim, por mais que o código `Lua` precise fazer, por exemplo, duas chamadas para adicionar um vértice e conectá-lo com outro, a agregação dos fluxos fará parecer que os dois acontecerem ao mesmo tempo.

5.3.5 DistributionPage

Este componente exibe as duas métricas coletadas como gráficos. Ele se atualiza das distribuições observando o fluxo de dados dado pelas classes `DegreeDistributionCollector` e `DistanceToRootDistributionCollector`. Ele utiliza a biblioteca `Recharts`[20].

Capítulo 6

Conclusão e trabalhos futuros

O desenvolvimento de tal ferramenta foi um desafio técnico por si só. As bibliotecas escolhidas, a arquitetura da aplicação e até mesmo a interface gráfica foram refeitas diversas vezes.

No final, chegamos a uma ferramenta, que, apesar de imperfeita, já é bastante útil para o dia a dia de um pesquisador na área de redes complexas, bem como estudantes do assunto.

A ferramenta já foi utilizada e testada por algumas pessoas com comentários positivos sobre ela.

Entretanto, como todo trabalho precisa ter um fim, algumas coisas ficam como melhorias para o futuro, mas sem tirar a utilidade da ferramenta atualmente.

6.1 Trabalhos futuros

Entre as coisas que ficaram como trabalhos futuros, acredito poder separar em 3 partes.

6.1.1 Correções Imediatas

Multi-Tarefa pode travar aplicação

Um dos maiores problemas que a aplicação tem agora é que a interface e a simulação são executadas na mesma `thread` com um modelo cooperativo de multi-tarefa. Isso faz com que seja fácil travar a aplicação inteira apenas escrevendo um laço infinito sem devolver controle para o simulador.

Para solucionar isto, uma maneira simples é utilizando `Web Workers`, que são uma forma para aplicações web separarem em *threads* diferentes o desenho da interface do processamento de outras tarefas.

Neste cenário, seria possível ter um temporizador que em caso da simulação

passar muito tempo sem responder, o simulador poderia interferir e finalizar (ou deixar o usuário decidir) a simulação.

Análise de Segurança

O código executado pelo simulador é inserido pelo usuário, que possivelmente pode ser fonte de ataques de terceiros. Isto porque não foi feita nenhuma limitação nas possibilidades do código `Lua` executado pelo `Fengari`. Isto significa que talvez, exista a possibilidade de executar algum código malicioso.

Enquanto o usuário for o único escrevendo o próprio código, isto provavelmente não seria um problema. Mas se imaginarmos um cenário onde um agente malicioso compartilha um código para um usuário executar o código, este modelo de ataque se torna relevante.

É necessário fazer uma análise deste modelo de ataque para poder mitigar riscos.

6.2 Generalizações

A aplicação, embora bastante genérica, ainda faz algumas presunções sobre o modelo que o usuário estará interessado. As duas principais generalizações que já tornariam o uso da aplicação mais abrangente são:

- Permitir grafos direcionados
- Coletores de métrricas programáveis (possivelmente em `Lua` também)

6.3 Integrações

O objetivo da aplicação era ajudar pesquisadores a entenderem modelos de redes. Uma das ideias iniciais era permitir o compartilhamento dos modelos, algo que não foi feito. Implementar integrações com provedores de armazenamento na nuvem ou um repositório centralizado para compartilhar e descobrir modelos pode ser uma adição boa para aumentar a colaboração entre pesquisadores.

Referências Bibliográficas

[1] BARABÁSI, A.-L., ALBERT, R. "Emergence of Scaling in Random Networks", *Science*, v. 286, n. 5439, pp. 509–512, 1999. ISSN: 0036-8075. doi: 10.1126/science.286.5439.509. Disponível em: <http://science.sciencemag.org/content/286/5439/509>.

[2] ERDŐS, P., RÉNYI, A. "On the Evolution of Random Graphs". In: *PUBLICATION OF THE MATHEMATICAL INSTITUTE OF THE HUNGARIAN ACADEMY OF SCIENCES*, pp. 17–61, 1960.

[3] ERDÖS, P., RÉNYI, A. "On Random Graphs I", *Publicationes Mathematicae Debrecen*, v. 6, pp. 290, 1959. Disponível em: <https://users.renyi.hu/~p_erdos/1959-11.pdf>.

[4] GILBERT, E. N. "Random Graphs", *Ann. Math. Statist.*, v. 30, n. 4, pp. 1141–1144, 12 1959. doi: 10.1214/aoms/1177706098. Disponível em: <https://doi.org/10.1214/aoms/1177706098>.

[5] AMORIM, B., FIGUEIREDO, D., IACOBELLI, G., et al. "Growing Networks Through Random Walks Without Restarts". In: *Complex Networks VII: Proceedings of the 7th Workshop on Complex Networks CompleNet 2016*, pp. 199–211, Cham, Springer International Publishing, 2016. ISBN: 978-3-319-30569-1. doi: 10.1007/978-3-319-30569-1_15. Disponível em: <https://doi.org/10.1007/978-3-319-30569-1_15>.

[6] FIGUEIREDO, D. R., IACOBELLI, G., OLIVEIRA, R. I., et al. "Building your path to escape from home", *arXiv e-prints*, art. arXiv:1709.10506, Sep 2017.

[7] "RxJS". Disponível em: <https://rxjs.dev/>.

[8] "ReactJS". Disponível em: <https://reactjs.org>.

[9] "D3.js". Disponível em: <https://d3js.org/>.

[10] HAGBERG, A. A., SCHULT, D. A., SWART, P. J. "Exploring Network Structure, Dynamics, and Function using NetworkX". In: Varoquaux, G., Vaught, T., Millman, J. (Eds.), *Proceedings of the 7th Python in Science Conference*, pp. 11 – 15, Pasadena, CA USA, 2008.

[11] PEIXOTO, T. P. "The graph-tool python library", *figshare*, 2014. doi: 10.6084/m9.figshare.1164194. Disponível em: <http://figshare.com/articles/graph_tool/1164194>.

[12] BASTIAN, M., HEYMANN, S., JACOMY, M. "Gephi: An Open Source Software for Exploring and Manipulating Networks". 2009. Disponível em: <http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154>.

[13] BATAGELJ, V., MRVAR, A. "Pajek— Analysis and Visualization of Large Networks". In: Mutzel, P., Jünger, M., Leipert, S. (Eds.), *Graph Drawing*, pp. 477–478, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN: 978-3-540-45848-7.

[14] SHANNON, P., MARKIEL, A., OZIER, O., et al. "Cytoscape: a software environment for integrated models of biomolecular interaction networks", *Genome Research*, v. 13, n. 11, pp. 2498–2504, nov. 2003. doi: 10.1101/gr.1239303.

[15] GANSNER, E. R., NORTH, S. C. "An open graph visualization system and its applications to software engineering", *SOFTWARE - PRACTICE AND EXPERIENCE*, v. 30, n. 11, pp. 1203–1233, 2000.

[16] LERUSALIMSCHY, R., FIGUEIREDO, L. H. D., FILHO, W. C. "Lua—An Extensible Extension Language", *Software: Practice and Experience*, v. 26, n. 6, pp. 635–652, 1996. doi: 10.1002/(sici)1097-024x(199606)26:6<635::aid-spe26>3.0.co;2-p.

[17] "TypeScript". Disponível em: <https://www.typescriptlang.org/>.

[18] "Fengari". Disponível em: <https://fengari.io/>.

[19] "Monaco Editor". Disponível em: <https://microsoft.github.io/monaco-editor/>.

[20] "Recharts". Disponível em: <http://recharts.org/>.

[21] IERUSALIMSCHY, R. *Programming in Lua, Second Edition*. Lua.Org, 2006. ISBN: 8590379825.

[22] “Lua 5.3 Reference Manual”. Disponível em: <http://www.lua.org/manual/5.3/>.

Apêndice A

Documentação da Ferramenta (em inglês)

Netmosa is a network analysis tools focused on network models. It allows you to program (in Lua) a model and see it unroll in real time.

A.1 Running a Model

To run a model in Netmosa, you first need to code in the code editor (or load an example and tweak the variables as described) and then you press **Run**.

Within the visualization screen (while the simulation is running) you can toggle the visualization mode (between Graph and Statistics) by clicking on the Chart Icon.

You can also export your file in GraphML format by clicking the Save Icon.

You can adjust the speed of the simulation by dragging the slider on the top bar.

A.2 Programming in Netmosa

We provide some examples you can check but ultimately you can write your own from scratch.

In order to be able to write your own models in Netmosa, you need to know th basics of Lua and get used to our standard library.

A.2.1 Our Graph Structure

In our graph, as usual, we have vertices and edges, but it also have vertex attributes.

Vertices are referenced by their Vertex ID (which is the index in the adjacency vector, so the first Vertex has Vertex ID 1, because Lua is 1-indexed).

Edges are just a list with two elements, which are the two Vertex IDs of the two connected vertices. Edges may also be referenced by an Edge ID

One special thing about vertices is that they can hold attributes, which are keyed by a string and contains a string value. One special attribute is color, that is used to define the fill color of the vertex when rendering it.

A.2.2 Learning Lua

There are plenty of lua resources out there, I'll list a few here:

- Official free online book: Programming in Lua[21]
- Official Reference Manual [22]

Caveats

We use a special implementation of Lua made in Javascript: Fengari, they claim it is 99% compatible with the PUC-Rio implementation, but there is a chance that something does not work properly. However, I haven't had any problems.

Also, you probably won't be using these standard libraries: *io*, *os*, and *debug*, so I'd focus on learning the **math** and **string** libraries, specially the former.

A.2.3 Standard Library

EdgeId and VertexId are just lua numbers, we are annotating them here to make it easy to understand what a function returns or receives.

```
addVertex(): VertexId
```

Adds a new vertex and returns its ID

```
connectVertices(id1: VertexId, id2: VertexId): EdgeId | Nil
```

Connect two vertices by their ids. If successfull, returns the edge id, otherwise returns nil.

```
setAttributes(id: VertexId, key: String, value: String): Boolean
```

Set attributes for a vertex, receiving the vertex index, the attribute name and value. One useful attribute is "color", which is used in the rendering. Returns wheter or not the attribute was set (it won't be set if the vertex doesn't exist).

```
getAttributes(id: VertexId, key: String): String | Nil
```

Gets the value of a previously set attribute. Returns nil if not set.

`getNeighbor(id: VertexId, neighborIndex: Number): VertexId | Nil`

Get the Id of a neighbor vertex, receiving the vertex Id and the neighbor index. If the vertex doesnt exist or the index is out of range, returns nil.

`getNeighborCount(id: VertexId): Number | Nil`

Get the number of neighbors a given vertex (by index) has. Useful for iterating through neighbors or getting a neighbor at random. If the vertex for the given id doesn't exist, returns nil

`getVertexCount(): Number`

Get the number of vertices the graph has. Useful for iterating through all vertices or getting one at random.

`getEdgeCount(): Number`

Get the number of edges the graph has. Useful for iterating through all edges or getting one at random.

`getEdge(id: EdgeId): (VertexId, VertexId) | Nil`

Get an edge by it's index. It is useful if you want to get the edge ends. It either returns two values (each VertexId associated with the edge) or it returns nil, if the edge doesn't exist.

`getRandomVertex(): VertexId | Nil`

Get a random vertex. Returns nil if there are no vertices.

`getRandomEdge(): (VertexId, VertexId) | Nil`

Get a random edge. It either returns two values (each VertexId associated with the edge) or it returns nil, if there are no edges.

`getRandomNeighbor(id: VertexId): VertexId | Nil`

Get a random neighbor given a vertex. It may return nil if there aren't any neighbors.