UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

Desenvolvimento de uma biblioteca paralela para cálculo de medidas de redes complexas

Sady Sell Neto

Desenvolvimento de uma biblioteca paralela para cálculo de medidas de redes complexas

Sady Sell Neto

Orientador: Francisco Aparecido Rodrigues

Monografia de conclusão de curso apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP - para obtenção do título de Bacharel em Ciências de Computação.

Área de Concentração: Coloque a área de concentração

USP – São Carlos
Novembro de 2015

Coloque a epígrafe nesta página se desejar.

Dedicatória

Dedico, em primeiro lugar, este trabalho a Deus, que com seu infinitos amor, bondade e misericórdia conduziu-me durante todo o período de graduação já cursado, incluindo o atual período de desenvolvimento do trabalho de conclusão de curso, assim concedendo-me a oportunidade de crescer nos âmbitos pessoal, intelectual e acadêmico. Dedico-o também a meus pais, as pessoas mais importantes para a realização do mesmo trabalho, devido aos seus auxílios financeiros, materiais, e, principalmente, psicológicos, sem diminuir a importância de seus conselhos para minha formação enquanto cidadão.

Agradecimentos

Agradeço ao meu amigo Loys Henrique Sacomano Gibertoni, com quem tive a oportunidade de morar quatro anos de minha vida, não só pela convivência, a qual contribuiu abundantemente para meu desenvolvimento como pessoa, mas também por todos os momentos partilhados, sejam eles felizes, tristes, de angústia, de confiança, de empolgação, de decepção; por todas as demais experiências e por sua valiosa amizade. Agradeço também ao professor Francisco Aparecido Rodrigues, o qual muito dedicou do seu tempo para mim, com o objetivo de ensinar, guiar minha caminhada na experiência que tive no meio acadêmico, em particular, nestes últimos meses, nos quais este trabalho foi desenvolvido; por incentivar os meus estudos de uma forma cada vez mais profunda, pela paciência mediante aos diversos contratempos e outros problemas, e também por sua amizade. Deixo meu agradecimento a todos os meus amigos que pude fazer nessa caminhada, bem como àqueles que eu já conhecia anteriormente a ela, por contribuírem, diretamente ou indiretamente, para minha formação, minha identidade, meu "ser quem sou hoje"; agradeço, especialmente, os que torceram por mim e incentivaram tal caminhada. Por fim, meus votos de gratidão aos meus irmãos na fé, os quais Deus me concedeu a grande alegria de conhecer em diferentes cidades e com cujas orações eu pude contar; conforto-me por saber que elas chegaram aos ouvidos do nosso Pai Celestial, por meio de seu Filho Jesus Cristo, fazendo com que Ele viesse em meu auxílio em minhas necessidades, com seu Santo Espírito sendo luz para conduzir meu caminho, sempre contando com a intercessão da bem-aventurada Virgem Maria, a quem agradeço com especial carinho e amor filial.

Resumo

Redes complexas são formadas por elementos conectados de forma não trivial, podendo ser representadas por grafos. Exemplos de redes complexas incluem a Internet, redes sociais e interações celulares. Embora essa área de pesquisa esteja bem desenvolvidas, há diversos desafios a serem superados ainda. Dentre esses desafios, o processamento de redes muito grandes, formadas por milhares, ou mesmo milhões de vértices, ainda é bastante limitado devido ao tempo computacional envolvido. Uma possível solução para essa limitação é o uso de recursos de processamento paralelo, como linguagens que usam passagem de mensagem. Assim, nesse trabalho propomos o desenvolvimento de rotinas para processamento de redes complexas com o uso de ferramentas de computação de alto desempenho, focando em sistemas distribuídos e programação concorrente. Nosso resultados, principalmente considerando o cálculo de medidas de distância, que são muito custosas, mostram que o uso dessas ferramentas é bastante promissor e pode ajudar no desenvolvimento da área de redes e análise de novas bases de dados que não podem ser processadas por métodos puramente sequenciais.

Palavras-chave: computação, sistemas, biblioteca, grafos, redes, complexas, concorrente, distribuídos, paralela, TCC.

Sumário

Lista de Figuras

Lista de Tabelas

CAPÍTULO 1: INTRODUÇÃO

1.1. Contextualização e Motivação

Sistemas complexos estão em toda parte, desde as interações celulares até as ligações entre pessoas nas redes sociais. Esses sistemas podem ser representados por redes complexas, cuja estrutura é matematicamente representada por grafos, ou seja, elementos discretos conectados de forma não trivial. O estudo de tais redes tem importância fundamental em áreas de vão desde a Biologia até Engenharia e Ciências da Computação. O estudo de redes complexas se baseia principalmente na sua caracterização, cujas propriedades topológicas são quantificadas por medidas, tais como medidas estatísticas e de distância . Assim sendo, diversas medidas têm sido desenvolvidas e usadas para descrever a estrutura da sociedade, as interações moleculares nas células, a organização do cérebro e as conexões entre roteadores na Internet.

Atualmente, existem algumas bibliotecas para se trabalhar com redes complexas, entre as quais duas se destacam: igraph [7] e NetworkX [10]. Porém, as rotinas por elas oferecidas operam de forma sequencial, utilizando apenas um núcleo de processamento. Por mais que tais rotinas permitam o processamento de redes, o cálculo de medidas mais complexas para redes de ordem de milhões de vértices torna-se inviável, restringindo a análise para o cálculo de medidas simples, ou então, o uso de aproximações numéricas. Dessa forma, a possibilidade de explorar sistemas distribuídos e programação concorrente para o cálculo das medidas mais custosas, ao reduzir a complexidade de tempo, elimina a inviabilidade de trata-las, dando nova luz a estudos e análises delas dependentes.

Este projeto em nada está associado com o desenvolvido para a disciplina anterior. Anteriormente, a disciplina cursada foi Projeto Supervisionado I em vez de Projeto de Graduação I, e contou com o tema “Serviços Web e Processamento Analítico de Dados”, o qual era um relatório de estágio, que descrevia a empresa, atividades desenvolvidas e aprendizados adquiridos, sendo que esses se concentram na área de desenvolvimento *full-stack* (*frontend*, *backend* e *mobile*), foco distinto do atual trabalho.

1.2. Objetivos

O objetivo principal do projeto consiste em desenvolver uma biblioteca pública, *open source*, que permita processar redes complexas e calcular medidas topológicas de forma paralela. Por biblioteca, entende-se um conjunto de funções pertencentes a um tipo abstrato de dados, sendo que esse modela computacionalmente um grafo ou uma rede complexa e aquelas a manipulam e realizam o cálculo de suas medidas, ambos capazes de serem importadas e utilizadas em outro código.

Várias são as funções de manipulação propostas a serem oferecidas, como adição de vértices, arestas, verificação da presença de arestas, entre outras. Quanto às medidas, a principal delas que será oferecida para ser calculada em paralelo são os menores caminhos, porém como existe uma série de outras medidas que dele dependem, ao paralelizar-se tal cálculo, seus dependentes, consequentemente, também o são.

Por se tratar de uma biblioteca, documentação também é um objetivo fundamental, para que o tipo abstrato de dados por ela exposto, bem como suas funções, sejam facilmente compreendidas e possam ser colocadas em uso sem dificuldades. Além disso, sendo *open source*, a documentação interna das funções, ou seja, do código que as funções executam, amplamente ajuda o futuro desenvolvimento dela por outros interessados.

1.3. Organização da Monografia

Conforme observamos anteriormente, o primeiro, fornece ao leitor a visão geral do trabalho e contextualiza-o, contribuindo na imersão dele para este projeto e auxiliando sua localização e disposição de elementos, com o objetivo de facilitar a localização mais precisa deles.

O capítulo 2 realiza uma revisão bibliográfica com o intuito de embasar qualquer discussão oriunda dele. Em tal capítulo, encontra-se a descrição dos conceitos que serão utilizados ao longo do trabalho, acrescentando a possível introdução e familiarização do leitor com a terminologia ao intuito do capítulo; segue-se, então, com a lista de trabalhos relacionados, os quais comportaram-se como modelos que guiaram o desenvolvimento deste. Finalmente, cita problemas, limitações e impossibilidades da versão atual e conjectura possíveis melhorias para versões futuras.

O próximo capítulo, o terceiro, é o central para este trabalho. Ele descreve o problema a ser resolvido por esse trabalho, ou seja, o quê exatamente ele propõe. Logo em seguida, é ricamente descrita a solução proposta e todas as atividades e passos que a compuseram. Discute-se, então, os resultados obtidos, juntamente com a metodologia para a obtenção deles e a validade dos mesmos.

O quarto e último capítulo é dotado de um viés crítico: primeiramente, sobre as lições aprendidas e desenvolvimento profissional alcançados com o trabalho em questão, sucedida por uma análise sobre o curso de graduações, composta tanto de aspectos positivos quanto negativos, e na qual são realizadas sugestões para ele.

Por fim, este trabalho conta com dois apêndices. No apêndice A, encontram-se imagens contendo o diagrama de classes de todo o projeto, os quais constituiriam um grande volume para serem incluídos no texto principal. O apêndice B descreve de modo detalhado dos parâmetros e pseudônimos de tipos encontrados na classe principal do trabalho e complementa a informação contida no diagrama de classes.

CAPÍTULO 2: REVISÃO BIBLIOGRÁFICA

2.1. Considerações Iniciais

Antes de descrever o trabalho propriamente dito, faz-se necessária a descrição de alguns conceitos, termos, trabalhos no qual o presente se baseia. Fazendo-o, torna-se possível uma descrição mais livre de quaisquer outros dados a ele relacionados, pois não há necessidade da preocupação com o não entendimento do seu conteúdo.

2.2. Conceitos e Técnicas Relevantes

A área de redes complexas é caracterizada por bastantes conceitos, devido a sua grande abrangência e diversidade de medidas criadas. Redes complexas são modeladas matematicamente como grafos. Logo, a fim de que a compreensão dos conceitos de redes complexas seja mais profunda, faz-se necessário entender alguns conceitos básicos da teoria dos grafos. Eles serão descritos à luz do livro Network Science [3] e do artigo Characterization of complex networks: A survey of measurements [11].

Dentre eles, o primeiro que merece ser citado é o conceito de grafo: trata-se de um modelo matemático que representa uma coleção de elementos interconectados, em que a informação não se encontra apenas nos elementos, mas também na conectividade entre eles. A figura 1 apresenta um grafo de exemplo, que modela os estados do Brasil, de forma que vértice, isto é, um elemento dele, representa um estado, e cada aresta, isto é, conexões entre os vértices / estados representa a existência de fronteira entre eles, ou seja, dois estados estarão conectados se fizerem fronteira um com outro. Esse exemplo possui propósitos unicamente demonstrativos e se preocupa em modelar somente os estados brasileiros e suas fronteiras, não refletindo informações como tamanho dos estados, orientação de suas fronteiras (norte, sul, leste, oeste e suas combinações), posição geográfica dos estados, entre outras.

Computacionalmente falando, a estrutura de dados que armazena os vértices e arestas é conhecida por “lista de adjacências”; no entanto, também há a abordagem matemática de “matriz de adjacências”, muito utilizada para se derivar outros conceitos e compreendê-los; do ponto de vista computacional, essa consome muito mais espaço que

aquela (exceto para grafos que não sejam esparsos, os quais são raros), inviabilizando seu uso.

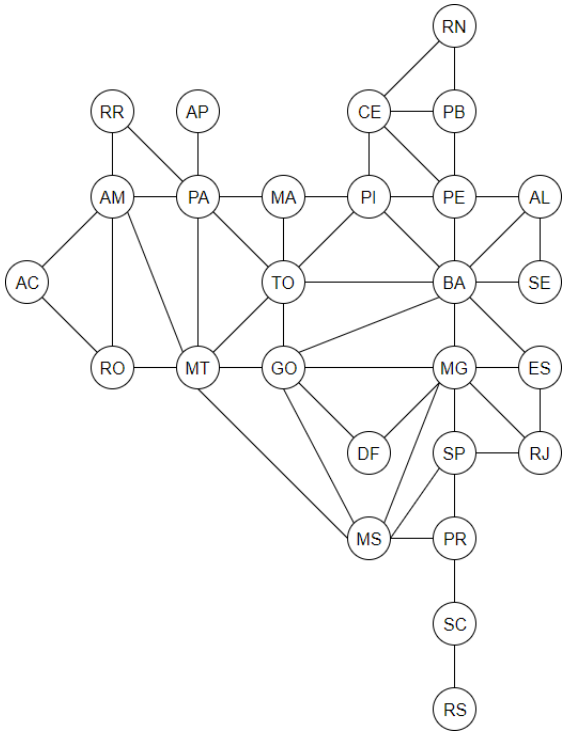

Figura 1: Grafo dos estados do Brasil e suas fronteiras

Sabendo-se que o grafo é composto de vértices e arestas, é natural querer saber o total de vértices e arestas. Ao primeiro dá-se o nome de ordem ou tamanho, já ao segundo muitas vezes não possui nome, mas algumas literaturas e bibliotecas, como o NetworkX [10], chama o total de arestas de tamanho, referindo-se ao número de vértices simplesmente como ordem.

A partir de vértices e arestas é possível definir um dos conceitos mais centrais de um grafo: o grau de um vértice, que é o número de arestas que um vértice possui. Por exemplo, no grafo exemplo representado na figura 1, a Bahia é o vértice com maior grau, possuindo arestas. Isso significa que ela é o estado brasileiro que mais faz fronteira com outros estados, fazendo-o com oito.

Outra pergunta que surge, eventualmente, olhando-se para figura 1, é: “Qual é o menor caminho entre dois estados?”. Naturalmente, essa modelagem que não considera a distância geográfica tampouco a área real dos estados, não poderá responder essa pergunta com precisão, fato esse que não invalida a pergunta. Porém, existe outra que, por mais

simples que o grafo apresentado seja, pode responder, que é: “Qual o caminho entre dois estados que menos passa por outros?”. O conceito por trás dessa pergunta e sua resposta é o conceito de menor caminho.

Para melhor compreender os menores caminhos, é prudente saber o que é um caminho. Caminho é uma sequência de vértices e arestas, começando e terminando em um vértice. Tendo ciência do significado de caminho, recorrendo à semântica da expressão “menores caminhos”, ter-se-á que o menor caminho entre dois vértices é um caminho, o qual possui ambos os vértices como extremos e que possui o menor número de arestas possível entre eles. Uma das formas para a obtenção de caminhos é por meio de travessias, as quais são algoritmos de visita aos vértices de um grafo a partir de um vértice fonte.

Existem várias medidas que usam os menores caminhos como base. Dentre elas, está a medida de centralidade de proximidade (*closeness centrality*), que define a centralidade do vértice como sendo inversamente proporcional à distância dele aos outros, de forma que o vértice mais central equivale ao que menos dista dos demais.

Em virtude dos grafos modelarem perfeitamente redes complexas, nomes alternativos para alguns conceitos supracitados foram sendo adotados, de maneira que tais nomes tivessem uma semântica maior: “rede” tornou-se sinônimo de grafo, “nó”, de vértice e “*link*”, de aresta.

De fato, redes complexas compreendem um conjunto deveras abrangente de conceitos. Entretanto, o trabalho conta com uma ferramenta fundamental para realizar o cálculo dessas medidas: sistemas distribuídos. Em verdade, mais do que apenas uma ferramenta, é uma verdadeira área da computação com conceitos próprios.

Começando pela própria expressão, um sistema de processamento distribuído é um sistema que utiliza vários computadores para o processamento de dados. Cada computador, o qual recebe o nome de “nó de processamento” ou “trabalhador”, inicia um processo para o cumprimento da tarefa designada. A máquina cujo processo dispara os demais, geralmente tida como o centro de processamento, é chamada de “mestre”, as demais, de “escravos”. A quantidade de dados que cada um precisa processar é comumente chamada de carga de trabalho, ou em inglês, *payload*.

Sistemas distribuídos ainda podem aliar-se a programação concorrente, no qual um processo cria subprocessos leves, chamados *threads*, para o processamento de um dado. Nesse modelo, tem-se paralelismo em nível de *thread* e naquele, em nível de processo.

Um dos principais intuitos de se recorrer a sistemas distribuídos, ou mesmo programação concorrente é o desejo de que um algoritmo seja executado em menor tempo real, devido à exploração do paralelismo. Para verificar o quanto isso aconteceu, existe uma medida denominada de *speedup*, que é calculada dividindo-se os tempo sequencial pelo tempo paralelo. Assim, a medida é diretamente proporcional ao tempo sequencial e inversamente ao paralelo, de forma que um baixo tempo paralelo (rapidez) resulta em um *speedup* maior.

Como o paralelismo de sistemas distribuídos acontece em nível de processo, é necessário que se estabeleça um mecanismo de comunicação entre processos (ou, em inglês, *I.P.C. – Inter-Process Communication*) para os processos distribuídos por entre os trabalhadores. Dentre as abordagens existentes, é digno de nota o modelo de passagem de mensagens, encapsulado pela interface de passagem de mensagens (em inglês, *M.P.I. – Message Passing Interface*), que estabelece um protocolo de comunicação entre os vários núcleos de processamento de um sistema distribuído; algumas implementações podem até considerar os núcleos como *threads*, explorando tanto sistemas distribuídos quanto programação concorrente em um único protocolo. Tal protocolo é um dos mais bem aclamados e possui bibliotecas que o implementam para diversas linguagens.

2.3. Trabalhos Relacionados

Como já comentado anteriormente, o livro Network Science [3] e o artigo Characterization of complex networks: A survey of measurements [11] são os textos dos quais formou-se a compreensão das medidas a serem desenvolvidas e listou-se o desenvolvimento das mesmas.

Também, segundo aquilo que já foi constatado, há uma diversidade de bibliotecas para se trabalhar com grafos, sendo que o igraph [7] e o NetworkX [10] são as mais conhecidas. Essa última, disponível apenas para Python, oferece um conjunto amplo de rotinas de manipulação de grafos, além de contar com funções de uso bastante intuitivo. Tal motivo fez com que se tornasse a principal fonte de referência para este trabalho:

bastantes decisões de projeto, principalmente quanto àquelas de como projetar a interface pública da biblioteca compreendida por este trabalho, assemelham-se àquelas oferecidas pelo NetworkX.

Quanto ao igraph, trata-se de uma biblioteca ainda mais completa que o NetworkX, além de estar disponível para mais linguagens: C/C++, Python e R; contudo, sua utilização (desde a instalação até o uso efetivo) não é tão simples, requerendo uma leitura mais minuciosa em seus tutoriais para que seja melhor compreendida. Devido a essas razões, tal biblioteca não foi a principal fonte para o trabalho realizado.

2.4. Considerações Finais

Nenhum trabalho é concebido espontaneamente; muitas são as fontes de conhecimento que a ele dão origem. A compreensão delas, sejam conceitos, terminologia, sejam outros trabalhos, se faz necessária, pois só assim pode tal trabalho ser compreendido em sua totalidade.

Logo, tendo plena ciência do contexto, expressões e contribuições para este trabalho, pode-se avançar na descrição do seu desenvolvimento.

CAPÍTULO 3: DESENVOLVIMENTO DO TRABALHO

3.1. Considerações Iniciais

Tendo descrito todo conhecimento necessário para que o trabalho em si seja compreendido, é chegado o momento de descrevê-lo. O trabalho, problema que ele resolve, algoritmos principais para solução, atividades realizadas, impactos, resultados obtidos serão agora listado. Também o serão limitações e problemas do trabalho.

3.2. Descrição do Problema

O principal objetivo desse trabalho é fornecer uma biblioteca, na linguagem C++, na forma de arquivos de cabeçalho, que seja capaz de calcular medidas de redes complexas e/ou grafos de forma paralela (explorando programação concorrente). A biblioteca será pública e *open source*, podendo ser encontrada em: https://github.com/DarkPaladin125/IC_Graph.

A escolha da linguagem C++ está no fato de ser uma linguagem que equilibra bem desempenho e modularidade, sendo uma das rápidas linguagens em que se pode desenvolver no paradigma orientado a objetos, paradigma esse que muito favorece o desenvolvimento de bibliotecas. Além disso, ela conta com estruturas de dados e algoritmos compreensivos e eficientes, na sua biblioteca padrão, a chamada STL [6]. Existe outra biblioteca formidável para essa linguagem, a Boost [1], que funciona particularmente bem juntamente com a STL e oferece algoritmos vastos para todos os propósitos, compondo um ferramental completo e rápido, que possibilita o desenvolvimento de bibliotecas para todos os fins, sem excluir o fim pretendido por este trabalho. Quanto ao uso de programação concorrente, trata-se de umas mais atuais e velozes ferramentas atuais no contexto de performance computacional, sendo ideal para realizar cálculos tidos como “computacionalmente caros”, como é o caso de redes complexas.

A arquitetura da biblioteca segue o diagrama de classes representado pela figura 2. O diagrama conta com apenas uma classe, a classe `graph` (além de suas classes aninhadas,

cujos diagramas podem ser examinados no apêndice A). Apesar de a classe ser grande, seu conteúdo é bem pertinente, configurando um cenário com boa coesão; o desmembramento em mais classes poderia diminuir a coesão ou aumentar o acoplamento, cenários indesejáveis em quase todo caso de arquitetura de *software*, ainda mais temíveis na concepção de uma biblioteca. Os apelidos para tipos (do inglês, *type aliases*) e parâmetros da classe são descritos no apêndice B; a presença deles se dá com o objetivo de que a classe se configure tal qual um contêiner da STL.

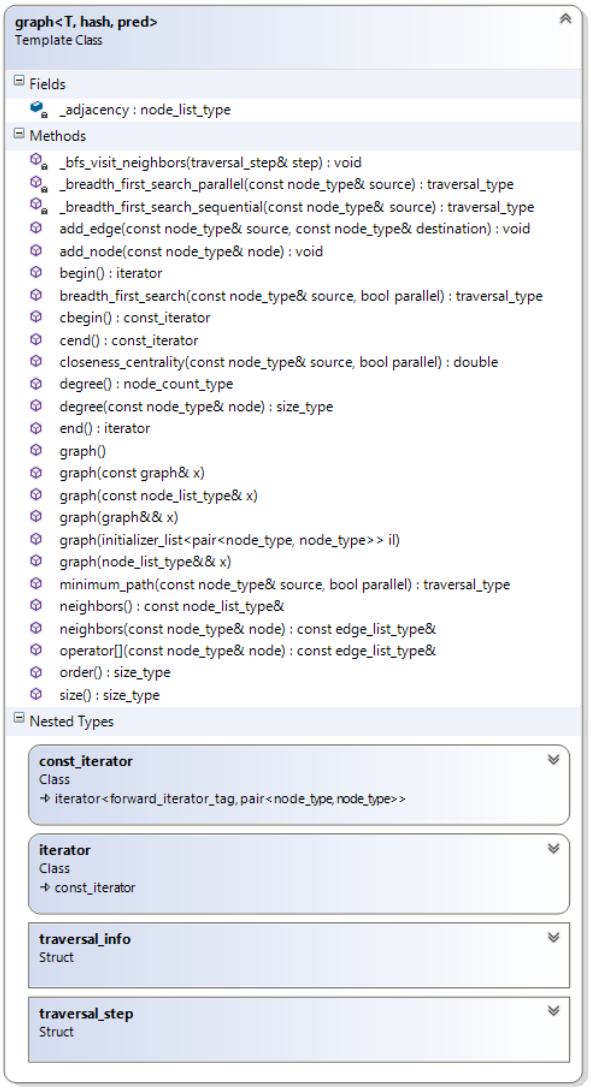

Figura 2: Diagrama de classes do projeto

Do diagrama de classes expressado pela figura 2, é possível ter-se uma ideia sólida da classe principal deste trabalho. Realizando uma análise no mesmo, nota-se, de imediato,

o grande número de sobrecargas de construtor, denotados por diferentes métodos com o nome da classe, pois existem diversas formas de se construir um grafo: o construtor sem parâmetros, que constrói um grafo sem vértices ou arestas; o que recebe uma lista de adjacências (as duas versões que estão presentes, uma em que possui o parâmetro é `const node_list_type&` e em que o mesmo é `node_list_type&&`, o estão por motivos recomendados pela STL para C++ versões 11 e superiores, pois têm um impacto profundo no desempenho), assumindo a lista fornecida como sua própria; o que recebe outro grafo (novamente, duas versões, pelo mesmo motivo daquele que recebe a lista de adjacências), copiando a lista do argumento para o novo grafo; por fim, o que recebe uma lista de inicialização, permitindo inicialização uniforme, fazendo que com que essa classe esteja em conformidade com os demais contêineres STL.

Para obter informações sobre o grafo, a classe conta com o método `order`, usado para se obter o número de vértices do grafo e o método `size`, para o número de arestas. A lista de adjacências, isto é, informações sobre a vizinhança entre vértices são fornecidas pelo método `neighbors`, que possui duas versões: a sem argumentos retorna, em um mapeamento, os vizinhos de todos os vértices, enquanto a com parâmetro, os vizinhos daquele especificado com argumento. Outra forma de se obter os vizinhos de certo vértice, é "indexar" um objeto com o mesmo vértice (obter os vizinhos do vértice `v` no grafo `g`: `g[v]`), operação possibilitada pela sobrecarga do operador `[]`, por isso o método `operator []`.

A manipulação da estrutura interna da lista de adjacências é dada por dois simples métodos: `add_node`, que adiciona um vértice recebido por parâmetro e `add_edge`, o qual adiciona uma aresta entre os vértices fornecidos como argumento.

Seguindo o padrão de projeto iterador, a classe oferece *forward iterators* [4], disponibilizados pelos métodos `begin` e `end`, e suas versões constantes, pelos métodos `cbegin` e `cend`, seguindo o mesmo padrão de nomenclatura da STL. Fazendo-o, torna-se possível colocar objetos dessa classe em cláusulas *range-based for* [5] da linguagem.

No coração da classe, expõem-se os métodos `minimum_path`, `breadth_first_search`, `_breadth_first_search_sequential` e `_breadth_first_search_parallel`, que correspondem ao verdadeiro núcleo do

trabalho. O primeiro deles calcula o caminho mínimo entre o vértice fonte, dado como o primeiro argumento e os demais outros para os quais é possível encontrar-se um caminho. O segundo argumento controla se o método deverá ser executado sequencialmente ou paralelamente, com o modo sequencial sendo o padrão, devido à configuração de *hardware* necessária para a execução do modo paralelo. Já o método `breadth_first_search`, realiza o algoritmo de travessia em largura (visita os vizinhos da fonte, depois os vizinhos dos vizinhos, e assim por diante), tendo como parâmetros os mesmos admitidos pelo método `minimum_path` e chamando os métodos privados `_breadth_first_search_sequential` e `_breadth_first_search_parallel` para verdadeiramente efetuar o processamento referente à travessia, de acordo com a opção de paralelismo. Vale notar que o método `minimum_path` é virtual; na classe atual, que modela um grafo sem peso, o caminho mínimo pode ser obtido por uma travessia em largura, fazendo com que a versão atual apenas propague a chamada para `breadth_first_search`, porém em casos com peso, podendo ser modelados com outras classes, as quais seriam subclasses da atual, deve ser obtido por outros algoritmos; sendo o método virtual, subclasses podem oferecer uma re-implementação particular e contemplar a mudança de algoritmo para caminhos mínimos.

Os métodos de travessia retornam um mapeamento de cada vértice em informações de travessia a ele relacionadas, expressadas por objetos da estrutura aninhada `traversal_info`, uma estrutura simples que armazena quem é o vértice antecessor na travessia e qual a distância do vértice fonte até ele. Ao se seguir os antecessores de dado vértice até o (vértice) fonte, é possível se construir o caminho da fonte até ele.

Tendo calculado os menores caminhos, é possível calcular uma gama de outras medidas, dentre elas o *closeness centrality*, implementado no método de mesmo nome, o qual tomando o vértice cuja centralidade se deseja saber como primeiro parâmetro e uma *flag* indicando se o paralelismo deve ser usado ou não, como segundo, calcula os menores caminhos e os utiliza para calcular a centralidade.

Já em questões de tecnologias de sistemas distribuídos, a biblioteca confia no modelo de passagem de mensagem, em particular, na implementação específica da MPI para a Boost [2]. O motivo da escolha está na familiaridade do autor deste trabalho com a

tecnologia mencionada, bem como a modularidade e clareza de código que a implementação citada oferece.

A proposta de modelagem da parte paralela do problema se enquadra na categoria SIMD (*Single Instruction, Multiple Data*) da taxonomia da de Flynn [12], utilizando o padrão *scatter-gather*, o qual, resumidamente, funciona da seguinte forma: um processo mestre divide (comumente divide-se igualmente) os dados a serem processados e manda em dispersão (*scatter*) uma porção de dados para cada escravo; cada trabalhador então realiza seu processamento e depois os dados são reunidos (*gather*) de volta no mestre para que ele os recomponha e realize eventuais processamentos sobre o dado recém-recomposto, como mostra a figura 3.

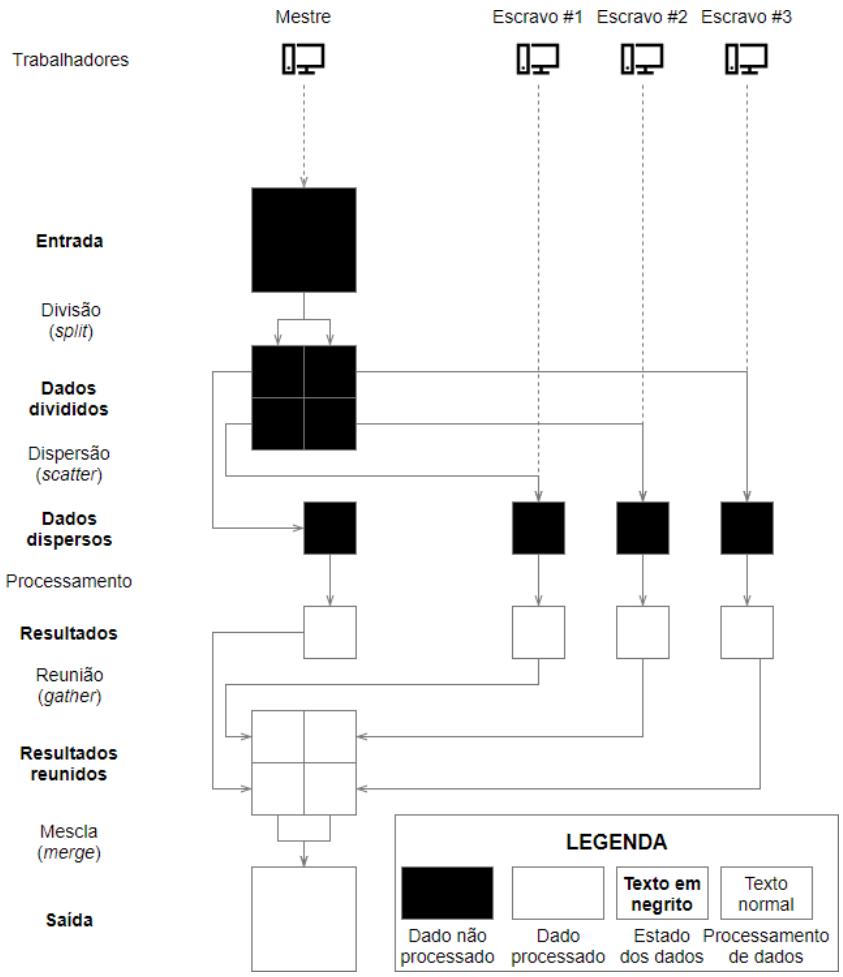

Figura 3: Cenário geral do padrão *scatter-gather*

3.3. Descrição das Atividades Realizadas

Para a concepção e implementação da biblioteca em questão, foram necessários estudos sobre redes complexas, teoria dos grafos e diversos testes sobre a eficiência, familiaridade e eficiência de uso sobre o contêiner subjacente. A partir desses estudos e experimentação, decidiu-se que para uma classe genérica / com *template* seria a abordagem mais intuitiva para um potencial usuário dela e, vinculada a tal decisão, a de fazer a estrutura de dados subjacente ser uma tabela *hash* associativa, pois, dado que o tipo de dados de um vértice não será conhecido de antemão (mas sim fornecido como argumento de *template*), uma estrutura de dados dispersa que armazena somente os vértices existentes de fato e que seja rápida seria ideal, características oferecida pela estrutura escolhida.

Terminadas as decisões de projetos, iniciou-se a implementação da classe `graph`. Em se tratando de um contêiner, existe uma miríade de métodos simples de acesso e manipulação que são necessários serem codificados para que o contêiner possa se comportar como tal. Tais são os diversos métodos que dizem respeito ao tamanho do grafo (juntamente com sua ordem) e consulta e manipulação da sua lista de adjacências. Outra característica para que a classe assemelhe-se a um contêiner da STL é a presença de iteradores. Ao passo que muitas classes contêineres podem ter implementações triviais para seus iteradores, ou seja, a manipulação deles é meramente alcançada obtendo iteradores para o contêiner subjacente, a classe graph requereu funções de manipulação própria, visto que o tipo sobre o qual se é iterado, o qual é um par de vértices (simbolizando uma aresta que os vincula), não é armazenado dessa forma no seu contêiner interno, fazendo-se presente entre as informações por ele armazenadas. Implementar iteradores não-triviais é uma tarefa igualmente não-trivial, sendo necessário dedicar uma fatia considerável de tempo para seu desenvolvimento e testes, sendo que esses foram realizados *ad hoc*.

Partiu-se, então, para mais uma etapa de estudos, dessa vez tendo como alvo conceitos de sistemas distribuídos, especialmente aqueles necessários para implementação em código do padrão *scatter-gather* proposto, incluindo estratégias para realizar as operações de divisão / *split* e mescla / *merge* dos dados de forma eficiente, atividade que precisou de um considerável tempo de dedicação.

Todavia, na última implementação, ocorrida antes da segunda etapa de estudos, já foi implementada a versão sequencial da travessia em largura, presente no método `_breadth_first_search_sequential`, chamado pelo `breadth_first_search`. Sua implementação é simples e, de fato, similar à fornecida em muitos livros e artigos de internet. O algoritmo utiliza apenas três estruturas de dados para realizar a travessia: a fila dos próximos vértices a serem visitados (implementado como um *deque*, para se manter a conformidade com a versão paralela), o conjunto de vértices já visitados e as informações de travessia já calculadas.

O algoritmo começa visitando os vizinhos do vértice fonte: as informações de travessia para eles são triviais, possuindo a fonte como antecessor e a distância de uma unidade, pois só foi necessário atravessar uma aresta da fonte até eles. Além disso, mantendo-se verdadeiro ao algoritmo de busca em largura, os vértices recém-visitados são enfileirados na fila de a serem visitados. A partir desse ponto, começa-se a versão iterativa do algoritmo: pega-se o próximo vértice a ser visitado e, caso de fato não tenha sido visitado (pode acontecer que ele tenha aparecido anteriormente na fila e já sido visitado), realiza-se o procedimento de visita: o antecessor do vizinho é o vértice atual, pois é por meio deste que aquele é acessado; similarmente, a distância do vértice fonte àquele é a distância a esse (que já é conhecida, pois se ele é o vértice atual, outrora foi visitado por algum outro, momento em que a distância dele foi calculado, com o processo eventualmente chegando aos vizinhos da fonte, que tiveram sua distância determinada fora do laço de iteração) mais um, a distância da aresta que os une; após isso, o processamento das informações de travessia daquele vizinho estão completas, e o vizinho deve ser visitado, entrando, portanto, na fila dos próximos; quando todos os vizinhos do vértice atual são visitados, o processamento do vértice atual é dado como terminado, podendo ser marcado como completamente visitado, entrando para o conjunto dos tais. A parte iterativa então se repete até que a fila de próximos esteja vazia, isto é, não haja mais nenhum vértice a se visitar.

Seguindo os estudos em sistemas distribuídos, finalmente pode ser concebido o algoritmo codificado no método `_breadth_first_search_parallel`, o principal de todo trabalho. Não se trata de um algoritmo trivial, sendo que até mesmo a estrutura `traversal_step`, que contém dados sobre um passo da travessia foi projetada para ele.

São quatro os seus membros: `immediate_visits`, uma fila (implementado como um deque, pois ele possui uma velocidade de serialização muito superior a da fila) de vértices a serem visitados na iteração atual do algoritmo; `future_visits`, uma fila de vértices a serem visitadas na próxima iteração; `visited`, um conjunto de vértices já visitados; e `info`, o mapeamento com as informações de travessia obtidas até o momento. Dois métodos são cruciais para o funcionamento dessa estrutura.

O primeiro é o método `split`, que divide a informação contida no objeto em tantas partes iguais quanto seu argumento mais um resto, o qual pode surgir caso a informação não seja perfeitamente divisível; a divisão, apenas divide o membro `immediate_visits`, copiando os membros `visited` e `info`, a fim de que, cada processo, quando receber uma parte da divisão, tenha a informação já computada inteira consigo, evitando que haja comunicação futura para obter essa mesma informação.

O segundo método é o método `merge`, que realiza a operação contrária do `split`: mescla, no objeto, as informações contidas no objeto parâmetro, concatenando o `future_vistis` (verificar quais elementos já estão presentes na fila é uma operação custosa; permitir o processamento múltiplo do mesmo vértice e devidamente tratar esse cenário é menos) do argumento no `immediate_visits` do objeto invocador (`this`), e realizando a união do conjunto `visited` e do mapeamento `info`. A classe também fornece uma versão estática desse método, o qual aceita um vetor de objetos dessa estrutura, com o objetivo de mesclar todos; para tal, ele cria um objeto vazio e vai realizando a mescla (não-estática) sobre cada elemento do vetor.

Quando o algoritmo começa, os vizinhos do vértice fonte são visitados sequencialmente, como na versão sequencial do algoritmo: vizinhos têm a fonte como antecessor e distância de uma unidade; vizinhos serão visitados, indo para `immediate_visits` e a fonte é inserida em `visited`, pois já está visitada. Inicia-se, então, a fase iterativa do algoritmo: é realizada a operação de divisão (*split*) e subsequente dispersão (*scatter*), para que cada trabalhador receba uma parte das visitas a serem realizadas. A parte restante, aquela que não pôde ser divida igualmente, é processada pelo mestre, assim evitando *overhead* na comunicação de rede. A forma com que a visita acontece é idêntico à versão sequencial: os vizinhos terão o vértice atual por antecessor e a distância de uma unidade a mais da distância do vértice atual, serão candidatos à visita na

próxima iteração, portanto passarão ao `future_visits` e o vértice atual está visitado, sendo inserido ao `visits`. Vale frisar que cada trabalhador executa essa tarefa de maneira independente dos demais, em seu próprio conjunto de dados. Terminado o processamento de cada trabalhador é realizada a operação de reunião (*gather*) e subsequente mescla (*merge*), centralizando toda informação no mestre. A etapa iterativa se encerra quando não houver mais nenhuma visita para iteração atual, isto é, membro `immediate_visits` vazio, caso em que a informação contida em `info` é o resultado final do algoritmo.

Como se trata, de fato, de um algoritmo complexo, a figura 4 ilustra um exemplo com o funcionamento do mesmo.

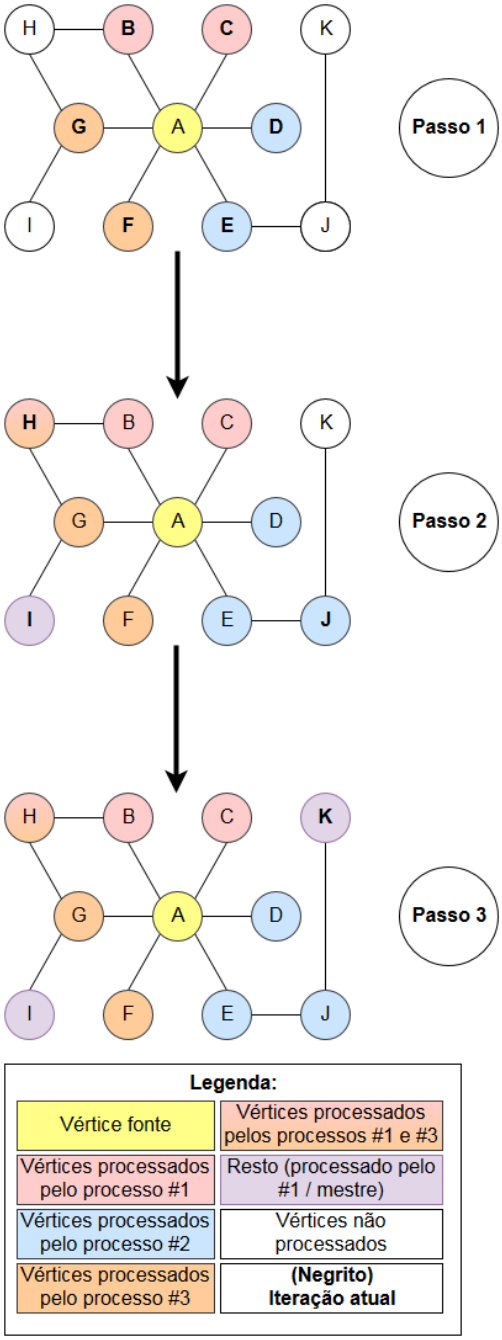

Figura 4: Exemplo da busca em largura em paralelo

Terminado esse núcleo da biblioteca, uma medida que usa informações provenientes dos menores caminhos (mais especificamente, a distância), o *closensses centrality* foi implementado.

Por fim, a última etapa, foi a documentação de todo código realizado, estado em que o código se encontra atualmente.

3.4. Resultados Obtidos

Para a obtenção dos resultados, várias redes foram utilizadas: algumas delas foram concebidas aleatoriamente sem parâmetro algum, ou seja, foram criadas à imaginação do testador. Outras foram obtidas a partir de modelos para geração de redes aleatórias, mais notavelmente, os modelos Erdős–Rényi (abreviado como modelo ER), Watts–Strogatz (abreviado como WS) e Barabási–Albert (abreviado com modelo BA), pelas funções de geração de grafos aleatórios oferecidas pela biblioteca NetworkX [10]. Por fim, redes reais obtidas do *website* Konnect [8] também estão incluídas nas bases de redes.

A validação foi realizada primeiramente das funções mais básicas de consulta e manipulação. Dado que, com exceção das redes obtidas no Konnect, as demais possuem um número de vértices e arestas fáceis de serem controlados, esses métodos são fáceis o suficiente de se validar com testes de mesa. Além disso, sua validade também foi confrontada com os resultados retornados pela biblioteca NetworkX.

Depois, começou-se o teste com os algoritmos de caminhos mínimos. A primeira a ser testada foi a versão sequencial do algoritmo, com seus resultados confrontados com os da biblioteca NetworkX. O tempo de execução do algoritmo sequencial também foi medido. Terminado ele, aplicaram-se testes à versão paralela, utilizando dois *notebooks* pessoais (isto é, não foram *clusters* / supercomputadores, *hardware* comum na utilização de sistemas distribuídos), ambos na mesma rede local, conectados à internet por cabo *ethernet*. Os testes foram bem-sucedidos e o tempo de execução também foi medido.

A tabela 1 mostra algumas amostragens de tempo para a rede real "Human Protein (Figeys)" do Konect [9]. Em verdade, ao se utilizar as funções da classe `graph`, uma rede dirigida tornar-se-á sem direção, devido ao tratamento que a classe dá para inserção de arestas. A tabela também exibe o *speedup* médio, considerando a seguinte fórmula para cálculo (considera a erro): $R = \frac{\langle S \rangle}{\langle P \rangle} \pm \frac{\langle S \rangle \sigma_P + \langle P \rangle \sigma_S}{\langle P \rangle^2}$, onde R é o *speedup*; $\langle S \rangle$ é a média dos tempos das execuções do algoritmo sequencial; $\langle P \rangle$ é a média dos tempos das execuções do algoritmo paralelo; σ_S é a o desvio padrão dos tempos das execuções do algoritmo sequencial; e σ_P é o desvio padrão dos tempos das execuções do algoritmo paralelo (todas as unidades usam o segundo com sua unidade).

Amostra / algoritmo	Sequencial	Paralelo
1	9,60249	1,75514
2	9,64774	1,73436
3	12,7669	2,37769
4	9,60617	1,71457
5	9,77429	1,76574
6	9,59506	1,7456
7	9,60053	1,76933
8	9,61638	1,75201
9	9,71378	1,74603
10	9,63196	2,19077
Média	9,970199	1,901624
Desvio padrão	0,986613	0,252343
Speedup	**5,24299 $\pm$ 1,21456**	

Tabela 1: Tempos de execução do algoritmo de busca em largura

O *speedup* consideravelmente alto mostra que o algoritmo proposto pelo trabalho é, de fato, mais rápido que muitos sequenciais propostos pela literatura atual. Tal melhoria é particularmente interessante para quem lida com redes complexas de um número elevado de vértices e arestas, casos em que o ganho do paralelismo se acentua, por exemplo, redes sociais.

Por fim, testes acerca do cálculo do *closeness centrality* foram realizados. O ganho no desempenho para esses métodos está justamente no uso de um cálculo eficiente para menores caminhos, fazendo com que a eficiência se propague.

3.5. Dificuldades, Limitações e Trabalhos Futuros

Várias foram as dificuldades encontradas para a confecção deste trabalho. Em primeiro lugar, como já citado, durante o desenvolvimento do projeto houve várias decisões de projeto a serem feitas, todas sobre as quais muita experimentação se fez necessária para a decisão definitiva.

Outra barreira enfrentada pelo trabalho coincide, de certa forma, com a solução por ele oferecida, que está justamente na tecnologia MPI. A instalação de suas bibliotecas e de seu ambiente de execução não é muito trivial e necessitou de vastas buscas para a configuração deles, em especial para integração com o compilador / IDE. Contudo, a instalação isolada da biblioteca não resolve imediatamente o problema: ainda existe a

limitação da necessidade de encontrar outro computador devidamente configurado para que se possa aproveitar a parte de sistemas distribuídos oferecida pela tecnologia.

Mais um problema digno de nota foi relativo às estruturas de dados presentes nos membros da estrutura `traversal_step`. Incialmente representadas como filas, o algoritmo estava apresentando um *speedup* inferior a 1, em outras palavras, a versão paralela estava sendo menos eficiente. Isso se dava por um problema de serialização nas filas. Ao se alterar para *deques*, o problema pode ser sanado e o impacto sobre o desempenho tornou-se visível.

Ainda que a classe `graph` possua um algoritmo caminhos mínimos paralelo eficiente, é limitada por apenas trabalhar com grafos sem peso, direção ou arestas múltiplas. Enquanto muitos problemas reais podem seguir essa modelagem, há aqueles que não podem, limitando o uso da biblioteca. Inclusive, para grafos ponderados, caminhos mínimos não são mais sinônimos de menores caminhos, requerendo outro algoritmo para o seu cálculo. Versões futuras da biblioteca podem incluir suporte a demais tipos de grafos, ainda que seja necessária uma reestruturação arquitetural, e dar suporte ao cálculo de mais medidas.

3.6. Considerações Finais

O trabalho vai se aproximando do final, com suas partes principais já expostas. O encerramento do trabalho, que se segue, é composto de opiniões pessoais do autor em relação ao curso que este trabalho conclui.

CAPÍTULO 4: CONCLUSÃO

4.1. Contribuições

O projeto de graduação expresso por este documento muito foi formativo e edificante na vida do autor. Em especial, ele complementou bem o projeto desenvolvido no semestre passado, o qual foi um projeto supervisionado em vez de um projeto de graduação. Essa combinação de experiências, que é a mais eminente das contribuições teve um efeito bastante benéfico, pois permitiu a experiências em ambas as possibilidades principais para depois da conclusão do curso: carreira acadêmica e mercado de trabalho, assim possibilitando o confronto e melhor julgamento sobre qual dentre elas é a que mais adequa à determinação, disposição e aptidão do aluno.

Este projeto também contribuiu amplamente para o crescimento acadêmico no que se tange à confecção de bibliotecas públicas, *open source*, controle de versão e assim por diante. Mais um aprendizado digno de nota é a evolução na habilidade de escrever relatórios, artigos e outras formas de publicação no meio acadêmico. Como outrora ocorreu o aprofundamento na vida do mercado de trabalho, incluindo rotinas de trabalho, expressões e jargões, comportamentos, por meio deste trabalho, adquiriu-se semelhante experiência relacionada ao meio acadêmico, incluindo o aspecto menos interessante da desanimadora burocracia excessiva envolvida.

Em geral, assim como o período de estágio, uma experiência muito positiva e marcante, de cujos frutos extraem-se benefícios por toda a carreira profissional.

4.2. Considerações sobre o Curso de Graduação

O curso de graduação é um curso excepcional. Suas disciplinas proporcionam uma base de conhecimentos sólida e bem-fundamentada, por meio da grande gama de conteúdos abordados e variedade de disciplinas oferecidas e conceitos ensinados com um nível de profundidade formidável. Os conceitos abordados permitem uma esplêndida carreira acadêmica e, sem eles, o profissional que deseja inserir-se no mercado de trabalho poderia encontrar sérias dificuldades para fazê-lo, e tampouco seria fácil aprender as tecnologias atuais para o desenvolvimento de projetos reais.

No entanto, muitas disciplinas são abordadas apenas em um âmbito teórico, oferecendo pouca ou nenhuma exploração da parte prática. Embora seja perfeitamente possível aprender a aplicação prática mapeando e adequando a teoria ao problema em questão, o pouco aprofundamento nessa área contribui para dificultar tal processo. O aluno que deseja se lançar no mercado de trabalho pode sentir-se completamente desnorteado ao se deparar com a amplitude e magnitude de um projeto real; pode sentir, também, dificuldade de realizar o mapeamento mencionado, ou seja, como aplicar e relacionar os tão enfocados conceitos teóricos em um caso prático real.

Naturalmente, o foco do curso não deve ser alterado totalmente para a área prática, pois sem a base teórica, não se é possível entender o uso de certa ferramenta ou tecnologia. Contudo, a introdução dessas em algumas disciplinas, a fim de que o aluno se familiarize também com situações reais seria bem-vinda. De fato, a tecnologia é inconstante: ferramentas que hoje são rotuladas como modernas podem passar rapidamente para a categoria do obsoleto, porém a ausência do contato com quaisquer ferramentas limita a visão de aplicação de conteúdos do aluno. Eventualmente, para tal, seria necessário reformular disciplinas, para que não abordem apenas conceitos teóricos; como exemplos, tem-se: Sistemas Operacionais, Gerência de Projetos, Redes de Computadores, Inteligência Artificial, entre outras. Não só a reformulação é relevante, mas a adição de disciplinas que tenham mais situações voltadas a projetos reais, ou então boas práticas aplicadas ao mercado de trabalho e afins também o são.

Academicamente falando, como já mencionado anteriormente, o curso é perfeito: aborda uma vastíssima gama de conceitos de computação, incluindo muitas, se não todas as suas áreas e explora cada um deles com grande profundidade, favorecendo a multidisciplinaridade, incentivando e incitando o desejo dos alunos evoluírem em seus conhecimentos em pesquisa.

REFERÊNCIAS

[1] ABRAHAMS, David; DAWES, Bernan; RIVERA, Renan. **Boost C++ Libraries** [S.l.]. Disponível em: <http://www.boost.org/>. Acesso em: 14 nov. 2017.

[2] ABRAHAMS, David; DAWES, Bernan; RIVERA, Renan. **Boost C++ Libraries**: MPI [S.l.]. Disponível em: <http://www.boost.org/doc/libs/1_65_1/doc/html/mpi.html>. Acesso em: 16 nov. 2017.

[3] BARABÁSI, Albert-Láslzó. **Network Science**. [S.l.]. Disponível em: <http://barabasi.com/networksciencebook/>. Acesso em: 09 nov. 2017.

[4] CPPREFRENCE. cppreference.com. **C++ concepts: ForwardIterator.** [S.l.] Disponível em: < http://en.cppreference.com/w/cpp/concept/ForwardIterator>. Acesso em: 15 nov. 2017.

[5] CPPREFERENCE. cppreference.com. **Range-based for loop** [S.l.]. Disponível em: < http://en.cppreference.com/w/cpp/language/range-for>. Acesso em: 15 nov. 2017.

[6] CPPREFRENCE. cppreference.com. **Standard template library.** [S.l.] Disponível em: <http://en.cppreference.com/w/cpp/header>. Acesso em: 14 nov. 2017.

[7] IGRAPH. The igraph core team. Disponível em: <http://igraph.org>. Acesso em: 09 nov. 2017.

[8] KUNEGIS, Jérôme. **KONNECT**: The Koblenz Network Collection. [Alemanha]. Disponível em: <http://konect.uni-koblenz.de/>. Acesso em: 18 nov. 2017.

[9] KUNEGIS, Jérôme. **Human protein (Figeys)**: Network analysis of Human protein (Figeys). [Alemanha]. Disponível em: <http://konect.uni-koblenz.de/networks/maayan-figeys>. Acesso em: 18 nov. 2017.

[10] NETWORKX. NetworkX. 2014. Disponível em: <https://networkx.github.io/>. Acesso em: 09 nov. 2017.

[11] RODRIGUES, Francisco Aparecido et al. **Characterization of complex networks: A survey of measurements**. Disponível em: <http://dx.doi.org/10.1080/00018730601170527>. Acesso em: 14 nov. 2017.

[12] TANENBAUM, Andrew Stuart. **Organização Estruturada de Computadores**. Rio de Janeiro, RJ: Livros Técnicos e Científicos Editora, 2001.

APÊNDICE A – Diagrama de Classes Completo

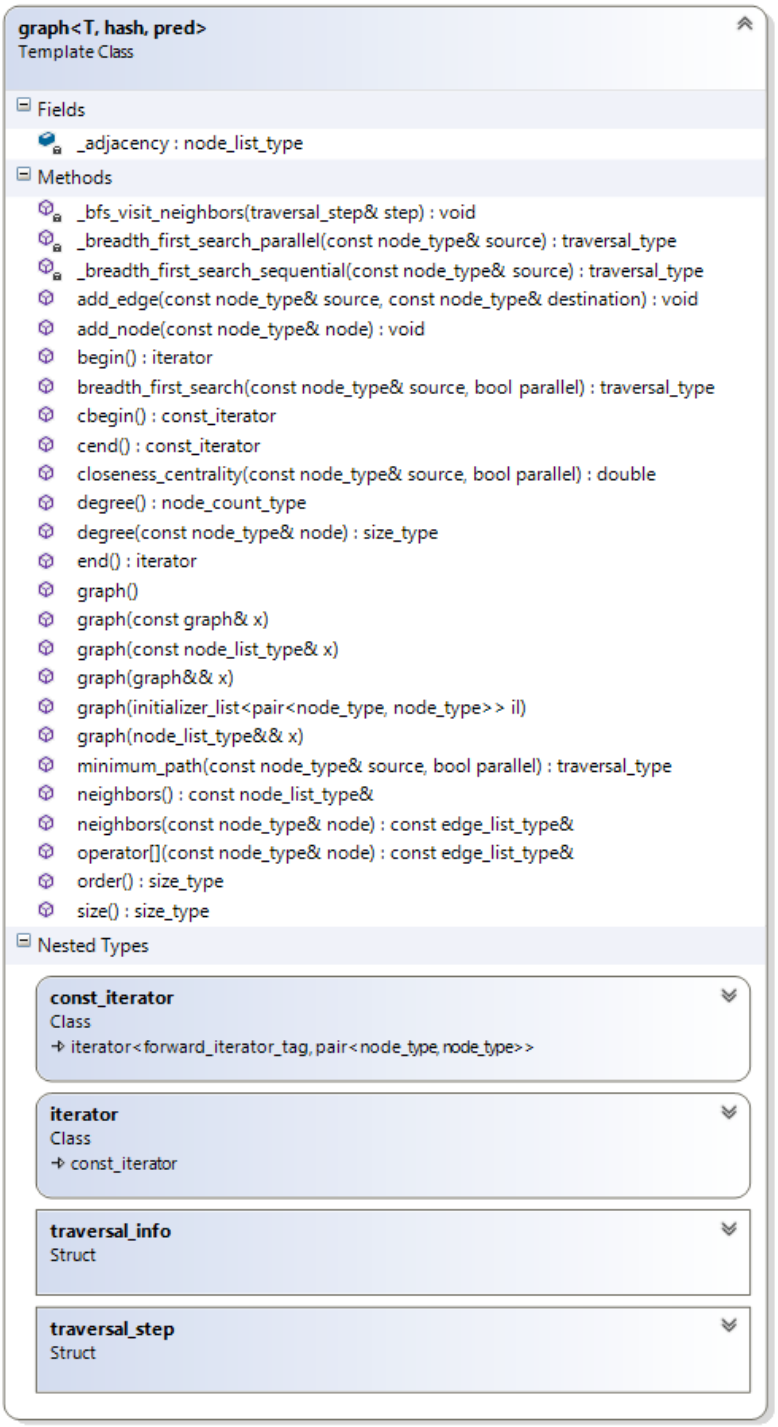

Figura 5: Diagrama de classes da classe `graph`

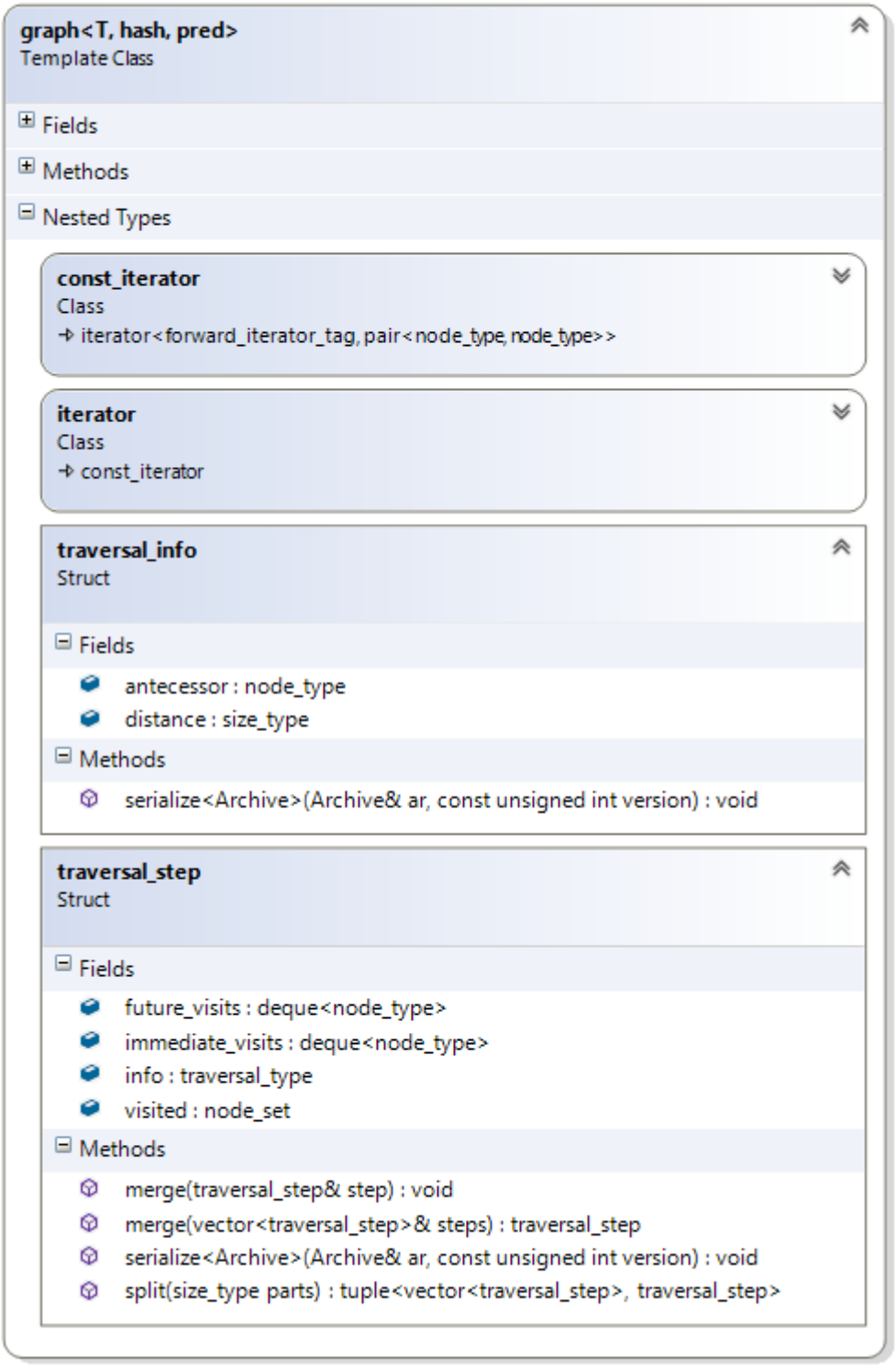

Figura 6: Diagrama de classes das classes aninhadas à classe `graph`

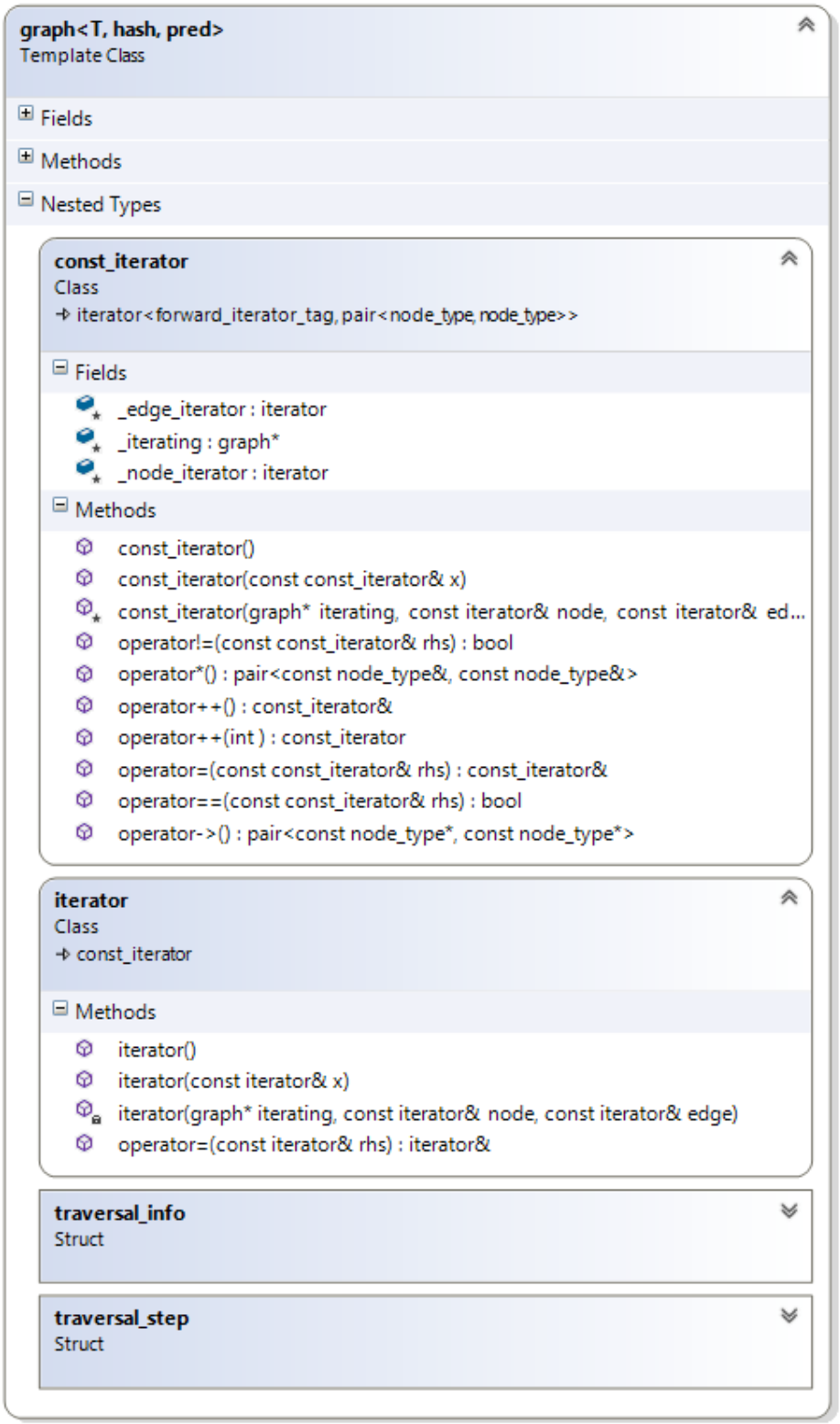

Figura 7: Diagrama de classes de iteradores da classe `graph`

APÊNDICE B – Parâmetros e apelidos da classe graph

A classe `graph` é uma classe parametrizada com *template*, em outras palavras, ela possui tipos que são configuráveis pelo usuário como argumentos. Da mesma forma que a função produz um resultado de acordo com seus argumentos, uma classe parametrizada necessita de argumentos, geralmente tipos de dados, a serem fornecidos para que possa dar sentido a seu conteúdo e comportamento.

Os parâmetros de *template* da classe `graph` são:

- `T`: o tipo de vértice que será armazenado no grafo, funcionando como "tipo chave". Em outras palavras, em um grafo de inteiros, os vértices são identificados por inteiros, enquanto em um de *strings*, por *strings*. É necessário que ele tenha um construtor de cópia (assim dito *copy constructible*) e tenha um operador de atribuição de cópia (assim dito *copy assingnable*);
- `hash`: função *hash* (de dispersão) que será usada para espalhar os dados, visto que a classe usa contêineres internos baseados em tabela *hash* para armazenar seus dados. Esse argumento aceita qualquer tipo chamável (como ponteiros para funções, classes que possuem o operador `()` / parênteses definido, o tipo declarado (obtido pelo operador `decltype`) de uma expressão lambda e outros tipos) que aceite um parâmetro do tipo `T` (o objeto a ser espalhado) e retorne um `size_t` (sua posição na tabela *hash*). Possui o valor padrão `std::hash<T>`, isto é, se o argumento não for fornecido, a função *hash* padrão da STL sobre o tipo do vértice será utilizada; para bastantes tipos padrão, oferecidos pela própria linguagem C++, e também para alguns tipos fornecidos pela STL, a função *hash* padrão é bastante eficiente;
- `pred`: função de comparação vara verificar se dois identificadores de vértices são iguais. Semelhantemente ao argumento `hash` supracitado, o argumento pode ser qualquer tipo chamável aceitando dois parâmetros do

tipo `T` (os objetos a serem comparados) e retornando um `bool` (`true` se iguais, `false` caso contrário). Possui o valor padrão `std::equal_to<T>`, ou seja, se omitido, será usado o operador `==` entre objetos do tipo do vértice para verificar sua igualdade (cenário seguro em muitos casos, pois são muitos os tipos, principalmente os tipos padrão, que contam com uma implementação bem definida para esse operador).

Outra observação importante sobre a classe `graph` é que ela define, em seu corpo, vários tipos aninhados. Alguns deles são tipos novos, com conteúdo próprio, como as classes `const_iterator`, `iterator`, `traversal_info` e `traversal_step`, enquanto outros são apelidos para tipos, gerando novos tipos sinônimos de outros. São eles:

- `node_type`: apelido para o argumento de template `T`;
- `hasher`: apelido para o argumento de template `hash`;
- `node_equal`: apelido para o argumento de template `pred`;
- `size_type`: apelido para `size_t`, o tipo mais comum para tamanho / contagem, pois tem grande abrangência e domínio de inteiros não-negativos apropriado para sua semântica.
- `edge_list_type`: tipo que representa, para cada vértice, os vizinhos com ele conectados, armazenando, assim, a sua lista de arestas; é uma apelido para `std::unordered_set<node_type, hasher, key_equal>`, o qual corresponde ao conjunto de vértices conectados a um determinado vértice;
- `node_list_type`: tipo que armazena a lista de vértices, cada um com sua lista de arestas, sendo o tipo de dados da lista de adjacências como um todo; é um apelido para `std::unordered_map<node_type, edge_list_type, hasher, key_equal>`, o qual corresponde a um mapeamento de cada vértice em sua lista de arestas / vizinhos particular.
- `node_set`: apelido para conjunto de nós, definido por questões de praticidade (diminui o tamanho do código produzido) e legibilidade (encapsula os argumentos menos comuns de *template*, que podem causar

confusão); é um apelido para: `std::unordered_set<node_type, hasher, node_equal>`;

- `node_map<Value>`: apelido para um mapeamento de um nó em algum outro definido por questões de praticidade (diminui o tamanho do código produzido) e legibilidade (encapsula os argumentos menos comuns de *template*, que podem causar confusão); é um apelido para: `std::unordered_map<node_type, Value, hasher, node_equal>`;
- `node_count_type`: mapeamento de cada nó em um tipo de contagem (um tipo inteiro não-negativo abrangente); é um apelido para `node_map<size_type>`, por sua vez, resolvendo-se em `std::unordered_map<node_type, size_type, hasher, node_equal>`;
- `traversal_type`: o tipo retornado por uma travessia, que corresponde a um mapeamento do nó nas informações de travessia com ele relacionados, notavelmente, antecessor e distância; é um apelido para `node_map<traversal_info>`, por sua vez, resolvendo-se em `std::unordered_map<node_type, traversal_info, hasher, node_equal>`.

$ Como adequar o conteúdo nas devidas seções, principalmente do capítulo 3?
$ Orientação das imagens... além disso, de repente, algumas poderiam se tornar anexos.
$ O resumo é aquilo?