CEFET/RJ
DIRETORIA DE PESQUISA E PÓS-GRADUAÇÃO

RIOGRAPHX: UM PORTAL CIENTÍFICO DE APOIO ÀS PESQUISAS EM TEORIA ESPECTRAL DE GRAFOS

Daniel Ferreira de Oliveira

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação, do Centro Federal de Educação Tecnológica Celso Suckow da Fonseca, CEFET/RJ, como parte dos requisitos necessários à obtenção do título de mestre.

Orientador(a): Leonardo Silva de Lima
Coorientador(a): Eduardo Bezerra da Silva

Rio de Janeiro,
Dezembro/2020

RIOGRAPHX: UM PORTAL CIENTÍFICO DE APOIO ÀS PESQUISAS EM TEORIA ESPECTRAL DE GRAFOS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação, do Centro Federal de Educação Tecnológica Celso Suckow da Fonseca, CEFET/RJ, como parte dos requisitos necessários à obtenção do título de mestre.

Daniel Ferreira de Oliveira

Banca Examinadora:

Presidente, Professor D.Sc. Leonardo Silva de Lima (UFPR) (Orientador(a))

Professor D.Sc. Eduardo Bezerra da Silva (CEFET/RJ) (Coorientador(a))

Professor D.Sc. Rafaelli de Carvalho Coutinho (CEFET/RJ)

Professor D.Sc. Virgínia Maria Rodrigues (UFGRS)

Professor D.Sc. Claudia Marcela Justel (IME)

Rio de Janeiro,

Dezembro/2020

Ficha catalográfica elaborada pela Biblioteca Central do CEFET/RJ

O48 Oliveira, Daniel Ferreira de
Riographx: um portal científico de apoio as pesquisas em teoria espectral de grafos / Daniel Ferreira de Oliveira — 2020.
80f. + anexo : il. color. , enc.

Dissertação (Mestrado) Centro Federal de Educação Tecnológica Celso Suckow da Fonseca , 2020.
Bibliografia : f. 75-80
Orientador: Leonardo Silva de Lima
Coorientador: Eduardo Bezerra da Silva

1. Teoria dos grafos. 2. Apache (Programa de computador). 3. Spark (Recurso eletrônico). 4. Framwork. 5. Análise espectral. I. Lima, Leonardo Silva de (Orient.). II. Silva, Eduardo Bezerra da (Coorient.). III. Título.

CDD 511.5

Elaborada pela bibliotecária Tania Mello – CRB/7 nº 5507/04

DEDICATÓRIA

Este trabalho de pesquisa é inteiramente dedicado aos meus pais Paulo e Glória e minha irmã Aline. Os três maiores incentivadores das realizações dos meus sonhos. Dedico também esta pesquisa a minha querida esposa Vanessa cuja presença sempre afetou positivamente a minha vida, em todos os aspectos.

AGRADECIMENTOS

Na realização da presente dissertação, contei com o apoio direto ou indireto de múltiplas pessoas e instituições às quais estou profundamente grata. Correndo o risco de injustamente não mencionar algum dos contributos quero deixar expresso os meus agradecimentos:

- Aos orientadores desta dissertação, os doutores Leonardo Silva de Lima e Eduardo Bezerra da Silva, pelas orientações prestadas, pelo seus incentivos, disponibilidades e apoios que sempre demonstraram. Aqui exprimo aos senhores toda a minha gratidão.
- Aos membros da banca por todas as considerações e críticas construtivas ao trabalho abordadas na defesa.
- A todos os amigos e colegas que de uma forma direta ou indireta, contribuíram, ou auxiliaram na elaboração do presente estudo, pela paciência, atenção e força que prestaram em momentos menos fáceis. Para não correr o risco de não enumerar algum não vou identificar ninguém, aqueles a quem este agradecimento se dirige sabê-lo-ão, desde já os meus agradecimentos.
- Não poderia deixar de agradecer à minha família por todo o apoio, pela força e pelo carinho que sempre me prestaram ao longo de toda a minha vida acadêmica, bem como, à elaboração da presente dissertação a qual sem o seu apoio teria sido impossível.
- À minha esposa por ter caminhado ao meu lado, pela sua paciência, compreensão e ajuda prestada durante a elaboração da presente dissertação, especialmente por apresentar sempre um sorriso, quando sacrificava os dias, as noites, os fins-de-semana e os feriados em prol da realização deste estudo

RESUMO

RioGraphX: um Portal científico de apoio às pesquisas em Teoria Espectral de Grafos

A Teoria Espectral de Grafos (TEG) é uma parte da matemática discreta que estuda as propriedades de um grafo a partir das informações fornecidas pelos autovalores e autovetores da matriz associada a este grafo. Esta teoria vem atraindo um maior interesse de pesquisadores desde a década de 80, em virtude da sua aplicação em diversas áreas, como na Química, na Matemática, na Engenharia e na Ciência da Computação. Com o crescimento exponencial do volume de dados a que se tem disponível atualmente, o processamento das informações em ambientes de execução de tarefas em paralelo e distribuído é crucial para uma melhor produtividade e desempenho. Com o objetivo de construir uma ferramenta WEB que dispensa o uso de recursos de processamento por parte do usuário, propomos o RioGraphX. Um portal científico desenvolvido utilizando o Apache Spark, que tem como objetivo obter todos os grafos que otimizam uma função matemática envolvendo invariantes de um grafo com possíveis restrições. Um *workflow* com sete etapas foi desenvolvido de modo a obter o máximo de tarefas possíveis executando no ambiente para computação paralela e distribuída do Apache Spark. Como o Spark fornece API para Scala, Java e Python, neste estudo foram desenvolvidos dois códigos-fontes: um na linguagem Java e outro em Python devido à abundância de bibliotecas de apoio. Em seguida, foram realizados dois testes: um de validação e outro de desempenho. A partir dos testes, cálculos de *speedup* e Eficiência compondo um comparativo de execução de tarefas em ambiente de processamento paralelo e distribuído com ambiente monoprocessado evidenciaram a superioridade do código desenvolvido em Java e a avaliação destas métricas de desempenho demonstram a importância da alocação dinâmica de recursos do Spark levando em consideração o tamanho da base de dados. Os tempos de execução do Portal se mostraram satisfatórios tendo em vista o volume de dados processados.

Palavras-chave: Teoria Espectral de Grafos; Apache Spark; RioGraphX; Portal científico

ABSTRACT

RioGraphX: a Portal to support research in Spectral Graph Theory

The TEG is a part of discrete mathematics that studies the properties of a graph from the information provided by the eigenvalues and eigenvectors of the matrix associated with this graph. This theory has attracted a greater interest from researchers since the 1980s, due to its application in several areas, such as in Chemistry, Mathematics, Engineering and Computer Science. With the exponential growth in the volume of data currently available, processing information in parallel and distributed task execution environments is crucial for better productivity and performance. In order to build a WEB tool that eliminates the need for processing resources by the user, we propose RioGraphX. A scientific portal developed using Apache Spark, which aims to obtain all graphs that optimize a mathematical function involving invariants of a graph with possible restrictions. A *workflow* with seven steps was developed in order to obtain as many tasks as possible running in the Apache Spark distributed and parallel computing environment. As Spark provides API for Scala, Java and Python, two sources were developed in this study: one in the Java language and the other in Python due to the abundance of support libraries. Then, two tests were performed: one for validation and the other for performance. From the tests, calculations of *speedup* and Efficiency composing a comparative of execution of tasks in parallel and distributed processing environment with monoprocessed environment showed the superiority of the code developed in Java and the evaluation of these performance metrics demonstrate the importance of dynamic allocation of Spark resources taking into account the size of the database. The execution times of the Portal were satisfactory considering the volume of data processed.

Keywords: Spectral Graph Theory; Apache Spark; RioGraphX; Scientific portal

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AGX	AutoGraphix
API	*Application Programming Interface*
ASCII	*American Standard Code for Information Interchange*
G6	*Graph6*
GAD	Grafo Acíclico Direcionado
HDFS	*HadoopDistributed File System*
JVM	Java Virtual Machine
NOSQL	*Not Only SQL*
RDD	*Resilient Distributed Datasets*
RDG	*Resilient Distributed Graphs*
SQL	*Structured Query Language*
TEG	Teoria Espectral De Grafos
UDF	*User-Defined Functions*
VNS	*Variable Neighborhood Search*

SUMÁRIO

Introdução

Contextualização

Os grafos, devido a sua natureza combinatória, servem de modelos para resolução de diversos problemas matemáticos. Estruturas que podem ser representadas por grafos estão em toda parte e muitos problemas de interesse prático podem ser formulados como questões sobre certos grafos.

A Teoria Espectral de Grafos (TEG) tem seu início atribuído ao trabalho de Hückel em 1931 na área de química quântica com o artigo [Hückel, 1931]. Neste trabalho, o autor representou a estrutura da molécula dos hidrocarbonetos não saturados como um grafo e obteve os níveis de energia de certos elétrons desta molécula a partir dos autovalores do grafo que modelava a molécula. Até meados da década de 80, centenas de trabalhos já haviam sido publicados por diversos pesquisadores como o de Hall [1970]. Época esta em que surge o livro *Spectra of Graphs: Theory and Application* [Cvetković et al., 1980], que resumiu quase todas as pesquisas até o momento na área. Esta teoria foi cada vez mais popularizada nas últimas duas décadas com Chung and Graham [1997].

A TEG é parte da matemática discreta que estuda as propriedades de um grafo a partir das informações fornecidas pelos espectros, ou seja, pelos autovalores e autovetores da matriz associada a um grafo. Como exemplo temos a matriz de adjacência, a laplaciana e a laplaciana sem sinal [Hogben, 2005]. Para certas famílias de grafos é possível caracterizar um grafo pelo espectro de uma destas matrizes que o representa, mas em geral isto não é possível. Os maiores e menores autovalores diferentes de zero dessas matrizes estão fortemente relacionados às propriedades estruturais do grafo. Por exemplo, pode-se citar o segundo menor autovalor da matriz laplaciana de um grafo, chamado de conectividade algébrica. Este parâmetro está associado à conectividade do grafo de modo que G é conexo se, e somente, a conectividade algébrica de G for positiva [Mohar et al., 1991].

De um modo geral, a razão do grande interesse na Teoria de Grafos é porque redes complexas, em vários ramos da ciência e da engenharia, tais como engenharia

elétrica, rede de computadores e moléculas bioquímicas podem ser modelados por grafos e suas propriedades podem ser obtidas por algoritmos computacionais. Na área da Ciência da Computação, o artigo de Cvetković and Simić [2011] apresentou contribuições da TEG em soluções para problemas em redes complexas e internet, em mineração de dados, em computação visual e reconhecimento de padrões, em buscas na internet, em balanceamento de carga de rede multiprocessadas, em proteções antivírus e em computação quântica.

Justificativa

Conjecturas da TEG podem obtidas desenhando pequenos grafos em papel, fazendo cálculos de invariantes à mão ou com calculadoras científicas, raciocinando sobre seus valores, depois realizando modificações nesses grafos e computando suas consequências. Esse processo pode ser auxiliado pelo computador, explorando a rapidez e agilidade para fazer cálculos e representar grafos da melhor maneira.

À medida que a ordem do grafo é incrementada, o número de grafos cresce muito rapidamente, onde mesmo nos sistemas computacionais modernos não é possível testar conjecturas ou estudar propriedades em todos os grafos onde o número de vértices é muito elevado. Por exemplo, a lista completa de grafos conexos com 14 vértices possui mais de $2,9x10^{16}$ grafos, de modo que a varredura exaustiva por todos esses grafos para o cálculo e análise de invariantes em tempo computacional razoável é inviável.

Ferramentas computacionais com o objetivo de auxiliar na proposição ou refutação de conjecturas e na descrição de famílias de grafos que satisfazem algumas propriedades foram desenvolvidas nos últimos anos. Uma ferramenta importante que permitiu a descoberta de diversas conjecturas é o Graffiti [Fajtlowicz, 1987]. A ideia básica do Graffiti é que a partir de uma lista de grafos conhecidos e presentes em sua base de dados, este é capaz de avaliar certas fórmulas a partir de invariantes. Se nenhum dos grafos com os quais o Graffiti está familiarizado for um contra-exemplo para uma fórmula, esta é considerada uma conjectura. Com o auxílio desta ferramenta, o autor propôs mais de 700 conjecturas [Fajtlowicz, 1998].

Outra importante ferramenta baseada na metaheurística *Variable Neighborhood*

Search (VNS) [Mladenović and Hansen, 1997] que tem sido usada para resolver muitos problemas em aberto na literatura é o AutoGraphiX [Caporossi and Hansen, 2000]. Em uma busca recente no Portal Science Direct utilizando o termo Autographix obtivemos o resultados de 78 artigos que citam ou resolvem conjecturas associadas a essa ferramenta.

Esse fato mostra como sistemas computacionais que auxiliam pesquisadores teóricos podem ser úteis para propor as conjecturas matemáticas corretas ou refutá-las. Outras ferramentas computacionais relevantes em TEG são Nauty-Traces and tools [McKay and Piperno, 2014], NewGraph [Brankov et al., 2006], MathChem [Vasilyev and Stevanović, 2014], Graph6Java [Ghebleh et al., 2019], House of Graphs [Brinkmann et al., 2013], MatLab [MATLAB, 2010] e o Graph Filter [Átila Arueira Jones and Ribeiro, 2020] . A Tabela 1 demonstra o quantitativo de citações que as principais ferramentas possuem, ressaltando a importância de ferramentas computacionais na produção de diversos trabalhos em TEG.

Ferramenta	**Citações**
Nauty-Traces and tools	916
Graffiti	358
AutoGraphiX	337
House of Graphs	156
Mathchem	19
NewGraph	10

Tabela 1 – Relação de registros de citações a cada ferramenta (fonte: Google Scholar. Pesquisa realizada em 10/11/2020)

Vale ressaltar que todas essas ferramentas dependem de recursos de processamento por parte do usuário, poucas possuem interface interativa e nenhuma delas está disponível para uso em um ambiente *on-line*. Além disso, a maioria requer algum nível de conhecimento em uma linguagem de programação específica e/ou utilização de sistemas operacionais baseados em Unix. Para verificar uma conjectura, pode-se codificar rotinas gerando todos os grafos para um determinado intervalo de vértices n e procurar por contra-exemplos, que é um processo demorado.

Objetivos

Segundo Curry et al. [2009], portais científicos têm se concentrado em fornecer aos cientistas acesso contínuo a conjuntos de dados e recursos computacionais. Ao abstrair os detalhes técnicos envolvidos, os cientistas são mais capazes de se concentrar em suas pesquisas sem precisar se tornar especialista em programação de sistemas.

Esta pesquisa apresenta o portal científico RioGraphX, composto por uma interface WEB e um *workflow* científico para execução de determinadas tarefas, cujo objetivo é fornecer à comunidade científica uma ferramenta computacional que dê suporte em pesquisas sobre TEG de modo a propor ou refutar uma conjectura associada a uma família de grafos que o usuário deseja verificar, através de uma análise exaustiva de todos os grafos de uma determinada ordem n. O RioGraphX apresenta o diferencial de ser uma ferramenta WEB que dispensa recursos de processamento por parte do usuário pois todos os cálculos são executados em um ambiente de processamento paralelo e/ou distribuído. A ferramenta aqui proposta provê uma interface WEB e amigável que facilita o manuseio para usuários menos experientes em informática.

Metodologia

Devido ao grande volume de grafos existentes e diante das limitações físicas impostas a processadores nos dias de hoje, técnicas de execução de algoritmos em paralelo e distribuído são necessárias para obtenção de rapidez e consistência nos resultados.

Com o advento do crescimento de pesquisas em ferramentas de processamento paralelo e distribuídos no processamento de grandes quantidades de dados pela comunidade científica, foi utilizado neste trabalho o Apache Spark [Karau et al., 2015], que é uma ferramenta de processamento de cargas de trabalho paralelo e/ou distribuído com o ganho de obter facilmente a integração com outros tipos de carga de dados como arquivos *batch*, algoritmos interativos e *streamings*.

De modo a abstrair ao usuário todas as etapas passíveis de automatização,

um *workflow* científico foi desenvolvido nesta pesquisa e apresenta sete etapas: a parametrização; a geração dos grafos que obedecem aos parâmetros da primeira etapa; o cálculo dos invariantes dos grafos gerados; a execução da função objetivo, informada na parametrização; a ordenação e filtragem dos grafos; e o relatório resumido de processamento do grafos, com as respectivas imagens. Todas as etapas foram implementadas no ambiente Spark, de modo a explorar ao máximo os seus principais recursos, e tem o auxílio de bibliotecas externas de apoio para manipulação dos dados.

Dois códigos-fontes foram desenvolvidos para estudo do *workflow* ideal: um utilizando a linguagem Python e o outro Java. Além disso, estruturas de dados do Spark foram analisadas de modo a obter melhor eficiência e agilidade no processamento dos resultados.

Após a construção dos códigos, foram realizados dois testes. O primeiro consistiu na validação dos resultados dos cálculos obtidos após o processamento do *workflow*. Escolhemos trabalhos na literatura onde os resultados são conhecidos e os reproduzimos nos dois códigos desenvolvidos. Ainda para comparação dos resultados, acessamos o ambiente CoCalc [SageMath, 2019] e reproduzimos os mesmos problemas.

O segundo teste foi elaborado em duas etapas. A primeira etapa consistiu em aferir as médias de tempos de execução entre os dois códigos de modo a verificar o mais rápido. Na segunda etapa foi o escolhido o código mais rápido, obtido na primeira etapa, e com ele foram realizados cálculos de *speedup*, que é a razão entre o tempo de execução em ambiente não-paralelo e em ambiente paralelo, e eficiência, que é a razão entre o valor obtido de *speedup* e o número de processadores utilizados, com diferentes configurações do Spark de modo a explorar a melhor forma de utilização de seus recursos.

Os resultados demonstraram que o código em Java é mais rápido à medida que a ordem dos grafos considerados é incrementada e que a utilização da capacidade máxima do Spark pode ser ineficiente de acordo com a quantidade de grafos carregados para processamento.

O Portal, fruto desta pesquisa, está disponível através do endereço `https://www.riographx.ga`.

Organização dos Capítulos

Esta pesquisa está organizada em sete capítulos. O Capítulo 1 introduz os principais conceitos de estudos da Teoria dos Grafos, algumas famílias de grafos, matrizes e invariantes importantes utilizados nesta pesquisa. O Capítulo 2 descreve as principais ferramentas desenvolvidas de apoio a estudos em TEG verificadas na comunidade científica e aponta algumas características delas em comparação com o Portal científico RioGraphX. O Capítulo 3 apresenta estudo do Apache Spark sobre sua arquitetura, suas funcionalidades e bibliotecas. O Capítulo 4 descreve toda a metodologia utilizada para composição desta pesquisa. No Capítulo 5 são descritos com mais detalhes as etapas definidas no *workflow* científico. No Capítulo 6 são abordados os testes de validação, os testes de desempenho de execução de cada *workflow*, cálculos de *speedup* e eficiência e avaliação do resultados. O Capítulo 7 descreve as conclusões obtidas e aponta trabalhos futuros. O Apêndice A apresenta um modelo de relatório em PDF gerado pelo portal científico e disponível para os usuários. O Apêndice B contém todos os invariantes disponíveis no portal e suas sintaxes de uso. E, por fim, o Apêndice C possui tabelas com todos os tempos de execução aferidos nas baterias de testes de desempenho.

1- Teoria dos Grafos

Neste capítulo são introduzidos os conceitos básicos relacionados a grafos que são utilizados ao longo desta dissertação. Apresentamos ainda representações de grafos em formas matriciais e definimos as notações com o intuito de facilitar o entendimento dos capítulos posteriores. A Seção 1.1 apresenta os conceitos básicos de teoria de grafos, algumas famílias de grafos e alguns invariantes que serão utilizados ao longo deste trabalho. Na Seção 1.2 são descritas as matrizes utilizadas para a representação dos grafos e invariantes derivados dessas matrizes.

1.1- Conceitos Básicos em Grafos

Um grafo não direcionado é definido como uma estrutura $G = (V, E)$, sendo V um conjunto finito não-vazio e E um conjunto de pares não-ordenados de V. Os elementos de V são denominados vértices e os de E são denominados arestas de G. O número de vértices (também conhecido como ordem do grafo) e o número de arestas de G são indicados, respectivamente, por $n = |V|$ e $m = |E|$. Cada aresta $e \in E$ é denotada por $e_{i,j} = (v_i, v_j)$, onde os vértices v_i e v_j são os extremos de e. Neste caso, como a aresta e é incidente a ambos os vértices, eles são chamados de vértices adjacentes. O número de arestas que incidem em um vértice v é chamado grau de v, denotado por $d(v)$. Arestas adjacentes são aquelas que possuem um vértice em comum.

A seguir, algumas definições e famílias de grafos que serão apresentadas para um melhor entendimento desta pesquisa.

Grafo simples, grafo valorado e grafo direcionado. Um grafo simples não apresenta laços e nem arestas múltiplas. Já um grafo valorado apresenta um valor (peso) associado a cada aresta que possui. E ainda, um grafo direcionado ou orientado apresenta arestas determinadas por pares ordenados. Nesse caso, a aresta (v_i, v_j) é representada por $(\overrightarrow{\mathrm{v_i v_j}})$. Esses grafos estão representados na Figura 1

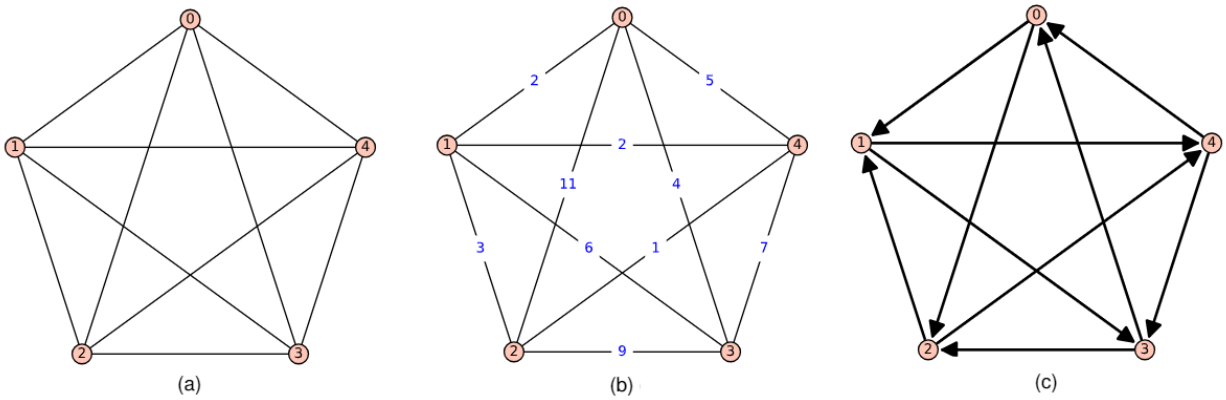


Figura 1 – (a) Grafo simples não-valorado; (b) Grafo simples valorado; (c) Grafo simples orientado

Caminho de um grafo. Um passeio de comprimento k em um grafo é uma sequência alternante de vértices e arestas $v_0, e_0, v_1, e_1, v_2, \ldots, v_{k-1}, e_{k-1}, v_k$, que começa e termina em vértices. Um caminho é um passeio na qual todos os vértices (exceto possivelmente o primeiro e o último) são distintos, enquanto uma trilha é um passeio no qual todas as arestas são distintas.

Um ciclo de comprimento $r \geq 3$ é um caminho constituído por $r + 1$ vértices, onde o primeiro vértice é igual ao último. Um caminho sem vértices repetidos é chamado de caminho simples e um ciclo sem vértices repetidos com exceção do inicial/final é um ciclo simples.

A distância entre dois vértices em um grafo é o comprimento do caminho mais curto entre eles, se houver, caso contrário, a distância é infinita. O diâmetro do grafo é a maior distância entre todos os pares de vértices do grafo.

Subgrafo. O subgrafo de um grafo é aquele em que o conjunto de seus vértices e o conjunto de suas arestas são, respectivamente, subconjuntos do conjunto de vértices e do conjunto de arestas do grafo dado. Ou seja, um grafo H é um subgrafo de um grafo G se, e somente se, $V(H) \subseteq V(G)$ e $E(H) \subseteq E(G)$. Desta forma, G é um supergrafo de H.

Grafo complementar. Um grafo $\overline{\mathrm{G}}$ é dito complementar de um grafo G se possuir a mesma ordem de G e (v_i, v_j) é uma aresta em $\overline{G}$ se, e somente se, (v_i, v_j) não é uma aresta em G.

Grafo conexo. Define-se G sendo um grafo conexo quando sempre existe um caminho entre cada par de seus vértices. Mas, se existir pelo menos um par de vértices em G que não verificam a condição anterior, o grafo G é denominado grafo desconexo. Um grafo G desconexo é composto pela união de componentes conexas. Na Figura 2, é apresentado um exemplo de um grafo conexo e um grafo desconexo com duas componentes conexas.

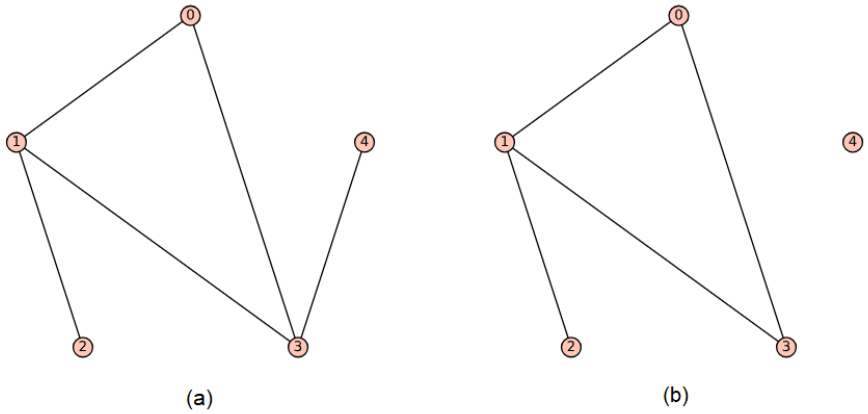


Figura 2 – Exemplo de um grafo conexo (a) e um grafo desconexo (b)

Grafo completo. Um grafo completo é um grafo simples de ordem n tal que cada vértice é adjacente a todos os outros vértices do grafo. Um grafo completo de ordem n tem $n(n-1)/2$ arestas e é designado por K_n. A Figura 3 demonstra grafos completos K_n para n = 1,2,3 e 4.

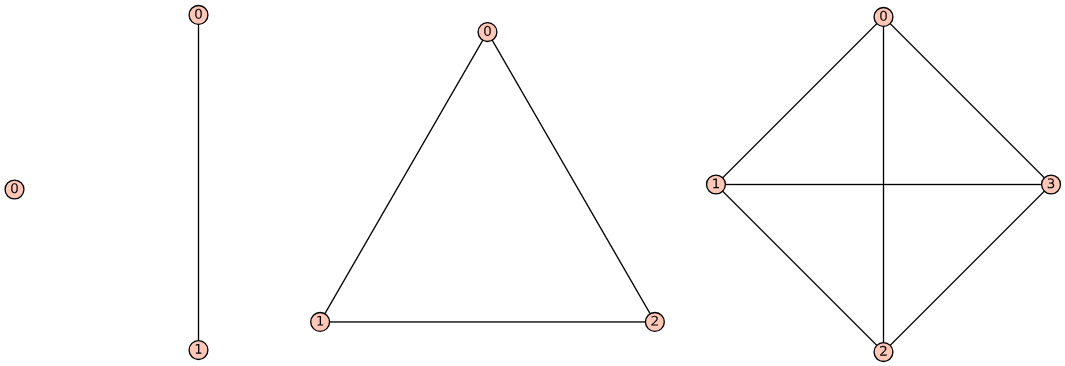


Figura 3 – Exemplo de grafos completos K_1, K_2, K_3 e K_4, respectivamente

Grafo bipartido. Um grafo é bipartido quando seu conjunto de vértices V pode ser particionado em dois subconjuntos disjuntos V_1 e V_2 tais que toda aresta do grafo tem uma extremidade em cada conjunto ou não existe aresta com extremidades no mesmo

conjunto. Um grafo é bipartido completo quando é bipartido e existem todas as arestas possíveis entre pares de vértices, conforme demonstrado na Figura 4.

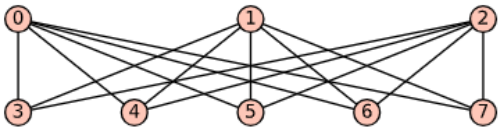


Figura 4 – Exemplo de grafo bipartido

Grafo livre de triângulos. Um grafo G é dito um grafo livre de triângulos quando G é um grafo não-direcionado no qual nenhum conjunto de três vértices forma um grafo K_3.

Coloração de um grafo. Se G é um grafo simples, a atribuição de cores para cada um dos vértices, de modo que vértices adjacentes apresentem cores diferentes, é uma coloração para G. O número cromático de G é o menor número de cores necessário para colorir G e é denotado por $\chi(G)$. Na Figura 5 é representada a coloração de um grafo com o seu respectivo número cromático $\chi(G)$

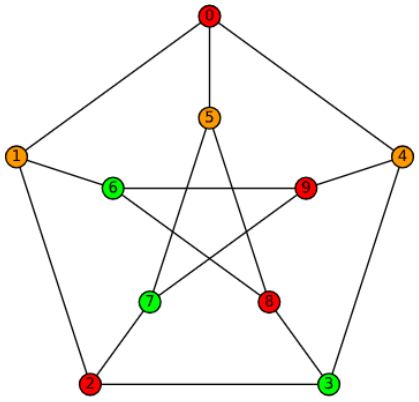


Figura 5 – Exemplo de coloração do Grafo de Petersen tal que $\chi(G) = 3$.

1.2- Matrizes associadas a grafos

Existem várias formas de representar grafos e o modelo computacional de estrutura de dados mais adequado é através de matrizes. Há diversos algoritmos de manipulação de grafos que utilizam vetores bidimensionais de números inteiros. Além

disso, muitas matrizes que representam grafos são naturalmente simétricas e há diversos resultados da Álgebra Linear que tornam esse tipo de representação bastante útil.

O estudo da TEG basicamente se caracteriza pelo relacionamento entre as propriedades algébricas do espectro de certas matrizes associadas a um grafo e as propriedades topológicas desse grafo.

Aqui nesta seção, apresentamos somente a definição das matrizes que serão abordadas nesse trabalho, ou seja, as matrizes de adjacência, laplaciana, laplaciana sem sinal, distância, distância laplaciana, distância laplaciana sem sinal e a ABC de um grafo G.

A. Matriz de adjacência

Seja $G = (V, E)$ um grafo simples e não direcionado com um conjunto de vértices $V = \{v_1, .., v_n\}$. A matriz de adjacência A é uma matriz simétrica e quadrada n x n, tal que

$$a_{ij} = \begin{cases} 1 & \text{, se } (v_i, v_j) \text{ é aresta em } G \\ 0 & \text{, caso contrário.} \end{cases}$$

Como exemplo, seja G o grafo simples da Figura 1a. A sua matriz de adjacência é representada da seguinte maneira:

$$A_G = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

O determinante det$(xI - A)$ da matriz de adjacência A é chamado de polinômio característico de G, denotado por $P_G(X)$. Os autovalores de A, os zeros de det$(xI - A)$, e o espectro de A (em que consiste os n autovalores) são chamados respectivamente de autovalores e espectros de G. Os autovalores de G são denotados por $\lambda_1, \lambda_2, \ldots, \lambda_n$ e eles são números reais pelo fato de A ser simétrica. Assume-se então que $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n$. É utilizada ainda a notação $\lambda_i = \lambda_i(G)$ para $i = 1, 2, \ldots, n$. O maior autovalor de G (λ_1) é chamado de índice de G.

Se λ é um autovalor de G, então um vetor não-nulo $x \in \mathbb{R}^n$, satisfazendo $Ax = \lambda x$, é chamado de autovetor de A (e de G) para λ; também chamado de λ-autovetor. A relação $Ax = \lambda x$ pode ser interpretada da seguinte maneira:

Se x = $(x_1, x_2, \ldots, x_n)^T$, então para qualquer vértice u tem-se $\lambda x_u = \sum_{v \sim u} x_v$, que é o somatório de todos os vizinhos v de u. Se λ é um autovalor de G, então o conjunto $\{x \in \mathbb{R}^n : Ax = \lambda x\}$ é um subespaço de $\mathbb{R}^n$, chamado de autoespaço de G para λ; sendo denotado por $\mathcal{E}(\lambda)$. Para o índice de G, pelo Teorema de *Perron-Frobenius*, todas as entradas do autovetor associado ao índice são entradas positivas.

Para exemplificar, seja

$$A_G = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

a matriz de adjacência A_G corresponde ao grafo completo K_4. Então :

$$P_G(\lambda) = \begin{vmatrix} \lambda & -1 & 0 & 0 \\ -1 & \lambda & -1 & 0 \\ 0 & -1 & \lambda & -1 \\ 0 & 0 & -1 & \lambda \end{vmatrix} = \lambda^4 - 3\lambda^2 + 1.$$

Os autovalores de G, obtidos como raízes do polinômio $P_G(\lambda)$, são $\frac{1+\sqrt{5}}{2} \approx 1.6180$, $\frac{-1+\sqrt{5}}{2} \approx 0.6180$, $\frac{1-\sqrt{5}}{2} \approx -0.6180$ e $\frac{-1-\sqrt{5}}{2} \approx -1.6180$.

B. Matriz laplaciana

Além do espectro da matriz de adjacência, considera-se também o espectro da matriz laplaciana de G, definida por $L_G = D_G - A_G$, onde D_G é a matriz diagonal dos graus dos vértices de G. Os autovalores da matriz L_G são todos não-negativos e o menor autovalor é sempre igual a zero; o segundo menor autovalor é também chamado de conectividade algébrica de G [Fiedler, 1973].

Sejam G um grafo com n vértices, D_G a matriz diagonal cujas entradas $D_{ii} = d_G(v_i)$

e A_G a matriz de adjacência de G. Define-se a matriz laplaciana de G como sendo:

$$L_G = D_G - A_G = \begin{cases} d_G(v_i) & \text{, se } v_i = v_j \\ -1 & \text{, se } (v_i, v_j) \text{ é aresta em } G \\ 0, & \text{, caso contrário} \end{cases}$$

Seja G o grafo simples da Figura 1a, a matriz laplaciana L_G de G é obtida da seguinte maneira:

$$L_G = D_G - A_G \begin{bmatrix} 4 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 4 \end{bmatrix} - \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 4 & -1 & -1 & -1 & -1 \\ -1 & 4 & -1 & -1 & -1 \\ -1 & -1 & 4 & -1 & -1 \\ -1 & -1 & -1 & 4 & -1 \\ -1 & -1 & -1 & -1 & 4 \end{bmatrix}$$

O espectro do laplaciano de G, denotado por $\zeta(G)$, é a matriz-linha cujos elementos são todos os autovalores de L_G ordenados em ordem não-crescente. Assim, se $\mu_1 \geq \ldots \geq \mu_n$ são os autovalores de L_G então $\zeta(G) = (\mu_1, \ldots, \mu_n)$.

O penúltimo autovalor do laplaciano de G, μ_{n-1}, é chamado conectividade algébrica do grafo G e é denotado $a(G)$. O maior autovalor do laplaciano de G, μ_1, é chamado índice do laplaciano do grafo G.

Se G é um grafo simples, então a soma dos elementos de cada linha da sua matriz laplaciana é igual a zero.

O polinômino característico de G da matriz laplaciana L_G é representado da seguinte maneira:

$$P_{L_G}(\mu) = det(\mu I - L_G)$$

Assim como a matriz de adjacência, L_G é uma matriz real e simétrica, portanto, todos seus autovalores são reais.

C. Matriz laplaciana sem sinal

Outro espectro bastante estudado atualmente é o da matriz laplaciana sem sinal (*signless laplacian matrix*), que é definida por $Q_G = D_G + A_G$.

Sejam G um grafo com n vértices, D_G a matriz diagonal cujas entradas D_{ii} = $d_g(vi)$ e

A_G a matriz de adjacência de G. Define-se a matriz laplaciana sem sinal de G como sendo:

$$Q_G = D_G - A_G = \begin{cases} d_G(v_i) & \text{, se } i = j \\ 1 & \text{, se } (v_i, v_j) \text{ é aresta em } G \\ 0, & \text{, caso contrário} \end{cases}$$

Sejam $q_1, q_2, \ldots, q_n$ os autovalores de Q_G com $q_1 \geq q_2 \geq \ldots \geq q_n$, o maior autovalor de q_1 é chamado de Q-índice de G.

D. Matriz de distância

A matriz de distância $Dist_G$ de um grafo conexo G é uma matriz simétrica n x n onde os elementos $Dist_{ij}$ são definidos como:

$$Dist_G = \begin{cases} l_{ij} & \text{, se } i \neq j \\ 0 & \text{, se } i = j \end{cases}$$

Onde l_{ij} é o comprimento do caminho mais curto, isto é, do número mínimo de arestas entre os vértices i e j de G. O comprimento l_{ij} é também conhecido como a distância entre os vértices i e j de G, consequentemente chamada de Matriz de distância de G [Mihalić et al., 1992]. Os autovalores desta matriz são definidos da seguinte forma : $\rho_1(G) \geq \rho_2(G) \geq ... \geq \rho_n(G)$.

E. Matriz de transmissão, de distância laplaciana e de distância laplaciana sem sinal

Para cada $u \in V(G)$, a transmissão de u em G, denotado por $Tr_G(u)$, é definida pela soma das distâncias de u para todos os demais vértices de G, ou seja, a soma da linha de $Dist_G$ indexada pelo vértice u. A matriz de transmissão de G (Tr_G) é matriz diagonal contendo os valores de transmissão de cada vértice de G.

A matriz de distância laplaciana [Aouchiche and Hansen, 2013] de um grafo G é definida como $D^L = Diag(Tr_G) - Dist_G$, onde $Diag(tr)$ é a matriz diagonal da matriz de transmissão de G. Os autovalores desta matriz são definidos da seguinte forma : $\gamma_1(G) \geq \gamma_2(G) \geq ... \geq \gamma_n(G)$.

De forma parecida, a matriz de distância laplaciana sem sinal [Aouchiche and Hansen, 2013] de um grafo G é definida como $D^L = Diag(Tr_G) + Dist_G$ e os autovalores

desta matriz são definidos da seguinte forma : $\alpha_1(G) \geq \alpha_2(G) \geq ... \geq \alpha_n(G)$.

F. Matriz ABC

Estrada [2017] apresentou a matriz ABC de G como a matriz quadrada $ABC(G) = (abc_{ij})$ onde os valores são definidos como:

$$abc_{ij}(G) = \begin{cases} \sqrt{\frac{d_u+d_v-2}{d_u d_v}}, \text{ se } (v_i, v_j) \text{ é aresta em } G \\ 0, \qquad\qquad \text{caso contrário} \end{cases}$$

Assume-se, de maneira geral, que seus autovalores estão ordenados de forma decrescente da seguinte forma: $\beta_1(G) \geq \beta_2(G) \ldots \geq \beta_n(G)$.

1.3- Outros invariantes

A seguir, são abordados demais invariantes disponíveis no Portal RioGraphX.

A. Energia de um grafo

A energia E_G de um grafo G [Gutman and Zhou, 2006] é definida pela soma dos valores absolutos dos seus autovalores. Consequentemente, seja A_G a matriz de adjacência de G e $\lambda_1, \lambda_2, ..., \lambda_n$ os autovalores de A_G, então E_G = $\sum_{i=1}^{n} |\lambda_i|$. A seguir apresentamos as definições das energias de outras matrizes associadas a grafos que estão implementadas no portal RioGraphX:

- Energia laplaciana de um grafo: $LE(G) = \sum_{i=1}^{n} |\mu_i - 2m/n|$
- Energia laplaciana sem sinal de um grafo: $SLE(G) = \sum_{i=1}^{n} |q_i - 2m/n|$
- Energia de distância de um grafo: $E_D(G) = \sum_{i=1}^{n} |\rho_i|$
- Energia de distância laplaciana de um grafo: $LE_D(G) = \sum_{i=1}^{n} \left|\gamma_i - \frac{1}{n}\sum_{i=1}^{n} D_i\right|$, onde D_i é a soma das distâncias entre v_i e os demais vértices de G, isto é, a i-ésima entrada da matriz de transmissão.
- Energia de distância laplaciana sem sinal de um grafo: $SLE_D(G) = \sum_{i=1}^{n} \left|\alpha_i - \frac{1}{n}\sum_{i=1}^{n} D_i\right|$, onde D_i é a soma das distâncias entre v_i e os demais vértices de G, isto é, a i-ésima entrada da matriz de transmissão.

B. Clique máxima

Uma clique em um grafo G é um subgrafo completo de G, ou seja, é um subconjunto $S \subset V(G)$ de vértices tal que todo par de nós em S está conectado por uma aresta. O tamanho da maior clique em G é denotado por $\omega(G)$, ou seja,

$$\omega(G) = \max_{S \subset V(G)} \{|S| \text{ tal que } G[S] \text{ é uma clique de } G\}.$$

O problema de encontrar a maior clique num grafo é NP-Completo.

C. Índice de conectividade *Atom-Bond*

O índice topológico é um valor numérico associado à constituição química para correlação da estrutura química com várias propriedades físicas, reatividade química ou atividade biológica. Existem algumas classes principais de índices topológicos, como índices topológicos baseados em distâncias, índices topológicos de conectividade e contagem de polinômios relacionados e índices de grafos. Estrada et al. [1998] propôs um índice topológico baseado na sequência de graus de um grafo, que é chamado de índice de conectividade *Atom-Bond* (ABC). O índice de conectividade *Atom-bond* de G é definido como:

$$ABC(G) = \sum_{uv \in E(G)} \sqrt{\frac{d_u + d_v - 2}{d_u d_v}}.$$

D. Índice Graovac-Ghorbani

Graovac and Ghorbani [2010] definiu um novo índice topológico inspirado em Estrada et al. [1998]. O índice de Graovac-Ghorbani é definido como:

$$ABC(G) = \sum_{uv \in E(G)} \sqrt{\frac{n_u + n_v - 2}{n_u n_v}},$$

onde n_u é o número de vértices com menor distância ao vértice u do que do vértice v na aresta $uv \in E(G)$. O valor de n_v é obtido analogamente.

2- Ferramentas Computacionais em TEG

Como o uso de ferramentas computacionais podem auxiliar as pesquisas teóricas em Teoria dos Grafos, e, em particular, nas pesquisas em TEG? Consideraremos uma resposta a essa questão e ainda apresentaremos as principais ferramentas computacionais utilizadas e mencionadas em artigos da área de TEG.

Para um dado grafo simples G com n vértices considere o conjunto G_n como o conjunto de todos os grafos simples de ordem n. Seja $f(G)$ uma função de invariantes do grafo G e considere as cotas superior e inferior desta função dadas por $u(G)$ e $l(G)$, ou seja,

$$l(G) \leq f(G) \leq u(G).$$

Dizemos que um grafo G é extremal em relação à função $f(G)$ se $f(G) = u(G)$ ou se $f(G) = l(G)$. Wilf [1967] determinou uma cota superior para o número cromático de um grafo, $\chi(G)$, em função do maior autovalor da matriz de adjacência do grafo, $\lambda_1(G)$, conforme apresentado no Teorema 2.1.

Teorema 2.1. *Seja G um grafo. Então,*

$$\chi(G) \leq 1 + \lambda_1(G).$$

A igualdade é válida se e, somente se, G é o grafo completo ou ciclo de comprimento ímpar.

Neste caso, veja que $f(G) = \chi(G),\ u(G) = 1+\lambda_1(G)$ e os grafos extremais são os grafos completos e ciclos ímpares. Em geral, é desejável determinar cotas de parâmetros combinatórios em função de autovalores. A justificativa é que existem algoritmos rápidos para a obtenção de autovalores e autovetores de matrizes simétricas, e estas cotas podem ajudar na aceleração de algoritmos para determinação de parâmetros combinatoriais com complexidade computacional na classe dos problemas NP, onde a existência de algoritmos polinomiais não é conhecida. Este é exatamente o caso de determinar o número cromático $\chi(G)$ de um grafo, pois esse problema é NP-Completo.

Um invariante conhecido na literatura é o *spread* de uma matriz, definido como a diferença entre o maior e o menor autovalores de um grafo. Assim, o *spread* da matriz laplaciana sem sinal de um grafo pode ser definida como:

$$s_Q(G) = q_1(G) - q_n(G).$$

A seguir apresentamos a Conjectura 2.1 proposta por Oliveira et al. [2010] ainda em aberto na literatura e que tem como grafos extremais casos especiais dos grafos caminho completo, denotados por $PC_{n,p,1}$, que são dados por $PC_{n,p,1} = \overline{K_{1,n-p-1} \cup pK_1}$.

Conjectura 2.1. *Seja G um grafo conexo qualquer com $n \geq 5$,*

$$s_Q(G) = q_1(G) - q_n(G) \leq \sqrt{4n^2 - 20n + 33}.$$

A igualdade é válida se e somente se G é isomorfo a um grafo caminho completo $PC_{n,1,1}$.

A verificação da Conjectura 2.1 de forma exaustiva para todos os grafos $G \in G_n$ com $n = 4, \ldots, 11$ pode ser uma boa estratégia para investir ou não tempo na prova do resultado. Com o uso de ferramentas computacionais, pode-se verificar esta conjectura do seguinte modo: considera-se a função

$$f(G) = q_1(G) - q_n(G) - \sqrt{4n^2 - 20n + 33}.$$

Se existe G tal que $f(G) > 0$, então G é contraexemplo para a conjectura e não há mais o que ser provado; caso, $f(G) \leq 0$ para todo $G \in G_n$ com $n = 4, \ldots, 11$, reafirma-se a motivação para que a prova da conjectura seja estudada.

Naturalmente, rotinas computacionais em linguagens como C++ e Python podem ser utilizadas para produzir algoritmos que geram todos os grafos de uma dada ordem n e testar as conjecturas desejadas. Entretanto, ferramentas dedicadas para esses tipos de problemas podem ser úteis no desenvolvimento dessas tarefas, o que é o objetivo dessa dissertação. Neste sentido, descrevemos a seguir um subconjunto de ferramentas computacionais utilizadas por pesquisadores da área de TEG e algumas de suas características. Em seguida, comparamos essas ferramentas e suas características com o portal RioGraphX proposto nesta dissertação.

A. Nauty-Traces and tools

O *Nauty-Traces*, cujos autores são Brendan McKay e Adolfo Piperno [McKay and Piperno, 2014], é um conjunto de ferramentas voltada para a geração de grafos e cálculo de invariantes. O *Nauty* utiliza uma árvore de busca para encontrar a representação canônica de um grafo. É a ferramenta mais difundida até hoje para o problema de isomorfismos de grafos. O Traces é uma evolução do *Nauty* no que diz respeito principalmente ao processo de busca em árvore. Se mostrou superior aos algoritmos concorrentes especialmente para grafos muito grandes.

O *Nauty-Traces and tools* consiste em um pacote de procedimentos implementados em C++ e podem ser executados em qualquer sistema operacional com um compilador C moderno, mas algumas aplicações precisam de recursos do tipo Unix. Isto é uma desvantagem do pacote, pois requer que usuários tenham conhecimento tanto em programação em linguagem C, como em uso de sistemas operacionais baseados em Unix. A vantagem é que o pacote possui um variedade de rotinas de exploração de diferentes famílias de grafos e são utilizadas em diversas pesquisas, inclusive esta.

Uma rotina existente na ferramenta e importante no contexto deste trabalho é a *geng* (do inglês, *graph generator*). Este procedimento gera todos os grafos simples não-isomorfos de uma dada ordem e os grava em um arquivo texto onde todos os grafos são representados num formato compacto chamado de *graph6* (g6). Este, criado por Brendan McKay, é um formato para descrever grafos usando caracteres *American Standard Code for Information Interchange* (ASCII) de forma bem simples e compacta. Seja um grafo G simples, ele começa com a metade superior de sua matriz de adjacência (sem uma diagonal zero), lista as colunas consecutivamente para obter uma matriz de bits e, em seguida, divide essa matriz em pedaços de seis bits cada (daí o nome graph6). Os números de seis bits obtidos desta maneira são adicionados a 63 bits para produzir caracteres ASCII visíveis (indo de '?' a '$\sim$' entre letras maiúsculas e minúsculas). O arquivo resultante consiste em linhas, uma para cada grafo, onde o primeiro caractere representa o número de vértices, também adicionado a 63 bits, e o os caracteres restantes na linha codificam a matriz de adjacência como descrito.

Como exemplo, seja G o grafo simples da Figura 1a e

$$A_G = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

a sua matriz de adjacência, a representação de G em g6 é feita pelo código *D*$\sim\{$.

B. NewGraph

A mais antiga das ferramentas desenvolvidas para estudos em TEG é um pacote interativo chamado GRAPH, que foi desenvolvido pela Universidade de Belgrado durante o período 1980-1984 [Cvetković, 1980].

O trabalho de Brankov et al. [2006] apresentou o NewGraph, uma evolução do GRAPH desenvolvida em Java que utiliza ferramentas mais sofisticadas para proporcionar um ambiente de trabalho mais confortável para o usuário. O NewGraph possui três *plugins* principais:

- *generators*, que usam os parâmetros fornecidos para criar um grafo particular de uma família específica de grafos;
- *invariants*, onde os invariantes selecionados são descritos em uma tabela abaixo do desenho do grafo e são automaticamente recalculados sempre que o grafo muda;
- *actions*, que realizam várias modificações no grafo.

Assim, com o NewGraph é possível desenhar o grafo desejado e computar diversos invariantes. A contribuição recente de [de Oliveira Ribeiro and Jones, 2020] realizou o desenvolvimento de diversas novas funcionalidades e *plugins* para o NewGraph, enriquecendo o software para o uso da sociedade acadêmica de estudos em TEG.

C. MathChem

O MathChem é um pacote Python de código aberto para o cálculo de índices topológicos e outras invariantes de grafos moleculares [Vasilyev and Stevanović, 2014]. Atualmente o pacote consiste em dois módulos: *MathChem* e *Utilites*

O módulo *MathChem* contém a classe *Mol*, que é a parte central do pacote. A classe *Mol* contém uma representação de um grafo molecular na forma de matriz de adjacência, juntamente com métodos para calcular vários invariantes e índices topológicos de grafos. O módulo *Utilites* contém um conjunto de funções para importação de grafos moleculares de arquivos externos para execução de processamento em lote em um conjunto de arquivos.

As principais funcionalidades do pacote são: cálculo de matrizes (de adjacência, incidência, laplaciana, laplaciana sem sinal, laplaciana normalizada e de distâncias), cálculo de propriedades espectrais de grafos (espectros de todas as matrizes, espectros de momento, energia), índices topológicos e cálculo de propriedade de grafos (ordem, diâmetro, grau, excentricidade, conectividade e distância entre vértices).

Por motivos de desempenho, o MathChem calcula invariantes de um objeto *Mol* sob demanda e, em seguida, salva os resultados para uso futuro. Todo objeto *Mol* tem seu próprio conjunto de variáveis privadas que é usado como um cache para essa finalidade. Deste modo, o MathChem evita o recálculo desnecessário de dados que consomem recursos, como matrizes ou suas propriedades espectrais.

Por fim, o MathChem possui integração com o Sage [Stein et al., 2008] e o NetworkX [Hagberg et al., 2008]

D. AutoGraphix(AGX)

O AGX foi introduzido por Caporossi and Hansen [2000]. A ideia principal do AGX é achar um grafo extremal para uma função que envolva um ou mais invariantes de grafos, possivelmente sujeitos a restrições. Desta forma, cada problema é visto como um problema de otimização combinatória da seguinte forma:

$$\max_{G \in G_n} f(G)$$

ou

$$\min_{G \in G_n} f(G).$$

Assim, o objetivo do AGX é determinar grafos que minimizam ou maximizam uma dada função dentre todos os grafos de ordem n. A busca sistemática por grafos que otimizam as funções acima é importante nos seguintes sentidos:

- Permite encontrar família de grafos que satisfaçam uma dada condição;

- Achar valores ótimos ou subótimos para um invariante sujeito a restrições;
- Refutar uma conjectura;
- Sugerir uma conjectura;

O AGX utiliza procedimentos baseados na metaheurística VNS [Mladenović and Hansen, 1997] para encontrar soluções de boa qualidade a partir de vizinhanças que realizam operações no grafo que alteram levemente a sua estrutura. Essas operações são:

- Remoção de uma ou mais arestas;
- Adição de arestas entre vértices não-adjacentes;
- Movimento de aresta, isto é, a remoção e adição de arestas mas nunca na mesma posição;
- Desvio, isto é, a remoção de uma aresta e adição de duas arestas em vértices intermediários de modo a unir os vértices órfãos da aresta removida;
- Atalho, operação reversa ao desvio;
- 2-opt, remoção de dois vértices não-adjacentes e adição de duas arestas diferentes entre os 4 vértices sobressalentes;
- Adição de vértice pendente;
- Remoção de vértice e, consequentemente, remoção de todas as arestas adjacente a ele.

A Figura 6 exibe as operações descritas anteriormente.

O AGX possui um fluxo de trabalho com três etapas definidas: definição do problema, grafo de partida e estratégia de busca. A definição da função objetivo com os invariantes a serem abordados e as restrições estão integradas como forma de penalidade durante as buscas pela vizinhança realizadas pela heurística baseada em VNS. Essas restrições são utilizadas na função objetivo também de modo a gerar soluções iniciais mais próximas do desejado, ao invés de grafos aleatórios. A próxima etapa do fluxo consiste em analisar o grafo, onde são realizadas as seguintes tarefas: executar rotinas de reconhecimento onde testes são feitos de modo a verificar se o grafo retornado pertence a uma classe conhecida; verificação dos invariantes, onde são retornados os principais invariantes do grafo selecionado; e visualização do grafo.

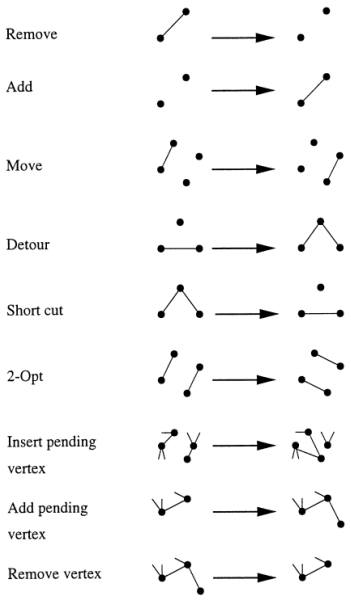


Figura 6 – Vizinhanças exploradas pela heurísticas implementada no AGX

Em 2017, foi publicado o artigo de Caporossi [2017] que apresentou o sistema AutoGraphix-III, ou seja, a terceira do versão do sistema com algumas melhorias significativas como melhorias na interface para o usuário final com mais informações disponíveis, melhoria no suporte para pesquisadores em redes complexas e otimização na heurística aplicado no sistema de maneira que o mesmo seja multiobjetivo.

Vale ressaltar que desde o desenvolvimento do AGX, em uma busca na base de dados do *ScienceDirect* foram encontrados de 78 artigos já foram publicados que mencionam a ferramenta, o que é um indicador de sua relevância na literatura. O AGX é a ferramenta que mais se assemelha à proposta nesta pesquisa.

E. Graph6Java

Ghebleh et al. [2019] desenvolveram uma ferramenta em linguagem Java para o auxílio de estudos de grafos para aplicações em química. A escolha pela plataforma Java se deu pela velocidade perante as outras linguagens testadas tendo em vista a vasta quantidade de bibliotecas de apoio, diminuindo a necessidade de muitos processamentos. Uma das principais bibliotecas utilizadas é a *colt* [Hoschek et al., 2004].

De forma a dispor uma interface mais simplificada para usuários menos experientes com a linguagem Java, é utilizada como apoio a ferramenta BlueJ [Kölling et al., 2003], que facilita a manipulação e execução dos códigos por aqueles usuários.

O primeiro passo do fluxo do sistema é definir qual das classes Java disponíveis deseja-se utilizar. São elas:

- Classe *Graph*, possui o método principal que cria o objeto de grafo com base na descrição contida no arquivo de entrada contendo grafos codificados em formato g6 e calcula os seus principais invariantes. Possui métodos simples que calculam o número de vértices e arestas de um determinado grafo e métodos mais complexos que calculam autovalores, autovetores, coespectralidade, integralidades e energia de Matrizes de Adjacência, Laplaciana, Laplaciana sem sinal e de Distância, bem como alguns índices importantes na literatura como os índices *Distance-sum heterogeneity* [Estrada and Vargas-Estrada, 2012], *Wiener* [Klavžar and Gutman, 1996], *Randic* [Randić, 1975] e *Zagreb* [Balaban et al., 1983] .
- Classe *ReporterTemplate*, uma classe auxiliar que carrega um determinado arquivo contendo grafos em formato g6, cria o objeto da classe Graph para cada grafo contido no arquivo e calcula uma função objetivo definida pelo usuário em linguagem java com base nos invariantes disponíveis na classe Graph;
- Classe *SubsetTemplate*, classe auxiliar que lista um subconjunto de grafos que satisfaçam uma dada condição, que pode ser de integralidade ou coespectralidade, a partir de um grafo dado como entrada;
- Classe *ExtremalTemplate*, na utilização desta classe auxiliar, um usuário seleciona um número definido de valores extremais (mínimo ou máximo) de um invariante também selecionado pelo usuário e a execução desta classe retorna todos os grafos que atendam as condições informadas pelo usuário;
- Classe *EquiTemplate*, lista um subconjunto de grafos que possuam (aproximadamente) valores iguais a um invariante selecionado de um conjunto de grafos.

Como resultado da execução da ferramenta, são gerados arquivos no formato .dot que podem ser utilizados no GraphViz [Ellson et al., 2001] para a geração da imagem dos grafos.

Os autores ainda demonstraram no artigo como o sistema auxilia na verificação de algumas conjecturas contidas na literatura de TEG.

A Tabela 2 apresenta um comparativo entre as características funcionais das ferramentas

abordadas nesta pesquisa.

Funcionalidades	Nauty-Traces	NewGraph	MathChem	AutoGraphiX	Graph6Java	RioGraphX
Manipulação de arquivos g6	Sim	Não	Sim	Sim	Sim	Sim
Interface interativa	Não	Sim	Não	Sim	Não	Não
Necessita noções de programação	Sim	Não	Sim	Não	Sim	Não
Necessita recursos de processamento local	Sim	Sim	Sim	Sim	Sim	Não
Acesso disponível 24x7 *on-line*	Não	Não	Não	Não	Não	Sim
Buscas por grafos extremais	Sim	Não	Não	Sim	Sim	Sim
Utiliza heurísticas	Não	Não	Não	Sim	Não	Não

Tabela 2 – Características funcionais de cada ferramenta abordada

A partir da Tabela 2, note que o portal científico RioGraphX tem a desvantagem de não possuir uma área interativa para desenho dos grafos e cálculo dos invariantes, se comparado ao AGX, por exemplo. Entretanto, a proposta principal da pesquisa não está focada nesta funcionalidade mas sim na geração exaustiva de grafos e sua ordenação em relação ao valor da função objetivo dada como entrada. Podemos ainda mencionar que uma característica do AGX é o uso da heurística VNS, ou seja, ao final de uma requisição o AGX fornece uma solução de boa qualildade, que não é necessariamente a solução ótima. Já no RioGraphX, ao final de uma execução, a solução ótima associada ao problema de entrada é obtida. As questões dos tempos computacionais e eficiência na geração de todos esses grafos com ordem até 11 vértices é um dos desafios desse trabalho e será tratada mais adiante.

3- Workflows científicos e o Apache Spark

Este capítulo apresenta o Apache Spark, ferramenta com estrutura fundamental para processamento de dados de forma paralela e distribuída, e alguns de seus componentes como Spark SQL, o GraphX e o GraphFrames. Componentes estes que foram objeto de estudo desta pesquisa de modo a dar suporte ao *workflow* científico desenvolvido.

Este capítulo está dividido em quatro seções. A Seção 3.1 descreve a definição de *workflow* científico. A seção 3.2 apresenta o Spark, sua arquitetura e como ocorre o seu funcionamento. A seção 3.3 descreve o Spark SQL, um componente utilizado para interações em formato *Structured Query Language* (SQL) com a *Application Programming Interface* (API) do Spark. Na Seção 3.4 é apresentado o GraphX, um componente voltado para processamento de grafos no Spark, e o GraphFrames, de propósito similar ao GraphX, que é uma ferramenta mais recente e com API que suporta mais interações com o Spark.

3.1- Workflow científico

Workflows científicos permitem que os usuários expressem facilmente tarefas computacionais de várias etapas, por exemplo, recuperar dados de um instrumento ou banco de dados, reformatar os dados e executar uma análise. Eles descrevem as dependências entre as tarefas e, na maioria dos casos, são descritos como um Grafo Acíclico Direcionado (GAD), onde os nós são tarefas e as arestas denotam as dependências da tarefa. As tarefas em um *Workflow* científico podem ser tudo, desde tarefas seriais curtas até tarefas paralelas muito grandes cercadas por um grande número de pequenas tarefas seriais usadas para pré e pós-processamento.

Como arquitetura base para construção do *workflow* científico definido nesta pesquisa, foram verificadas algumas bibliotecas que potencializassem o processamento paralelo das diversas tarefas a serem executadas. Bibliotecas como OpenMP/MPI, Paral-

lel BGL [Gregor and Lumsdaine, 2005], Ray [Moritz et al., 2018] e Hadoop MapReduce [Dittrich and Quiané-Ruiz, 2012] foram verificadas, mas estudos do Apache Spark como o de Mavridis and Karatza [2017] e Shi et al. [2015] demonstram que este possui desempenhos melhores, controle de falhas e melhor controle sobre paralelismo entre as tarefas iterativas.

3.2- Apache Spark

Segundo Karau et al. [2015], o projeto de pesquisa da ferramenta Spark foi iniciado pela Universidade da Califórnia, em Berkeley, onde a equipe vinha trabalhando no desenvolvimento da ferramenta MapReduce [Dean and Ghemawat, 2008] mas observaram que esta se mostrou ineficiente em alguns casos.

De início, a ferramenta Spark foi desenvolvida para interações rápidas entre algoritmos, mas trouxe boas ideias por seu armazenamento em memória e possuir recuperação a falhas. A velocidade de obter informações em grandes *datasets* é um desafio hoje em dia com processamento de grande demanda de quantidade de dados, *streamings* e consultas interativas. Diante deste cenário, a ferramenta Spark oferece um ganho maior por seu processamento de memória eficiente e sua arquitetura é facilmente acessível oferecendo uma simples API para acesso às suas funcionalidades.

Ela foi desenvolvida para receber uma carga de trabalho bem extensa e de diferentes processos de trabalho que requerem sistemas paralelos e/ou distribuídos. Por conta dessa vantagem, é capaz de combinar diferentes tipos de dados como arquivos em *batch*, algoritmos interativos e *streamings*. De maneira geral ele simplifica a manutenção em manter diversas ferramentas em um sistema.

Conforme demonstrada na Figura 7, integram a ferramenta os componentes Spark SQL, para trabalhos com dados estruturados; Spark Streaming, que permite o processamento ao vivo de *streaming*; MLlib Machine Learning, que inclui uma biblioteca vasta de algoritmos de aprendizado de máquina; e o GraphX, que possui biblioteca para manipulação e processamento paralelo de grafos.

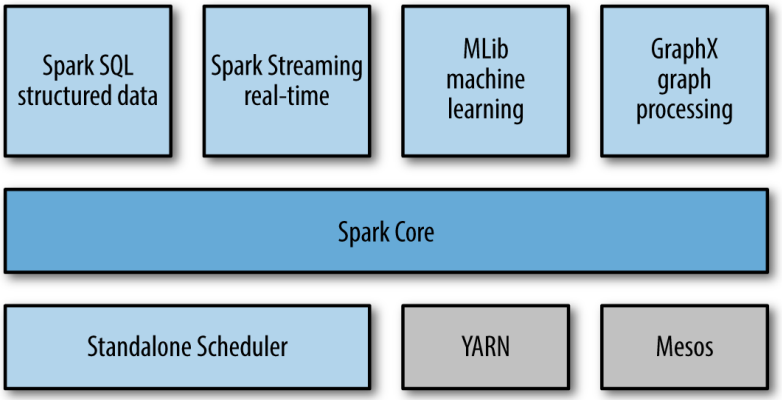


Figura 7 – A arquiterura Spark composta pelos componentes SparkSQL, Spartk Streaming, MLlib e o GraphX. (Fonte: http://spark.apache.org/)

3.2.1 Arquitetura do Spark

A arquitetura de funcionamento do Spark é constituída por três componentes principais, conforme demonstrada pela Figura 8:

- O *Driver Program*, que é a aplicação principal que gerencia o ambiente e é quem executará o processamento definido pelo algoritmo do sistema a ser executado;
- O *Cluster Manager* é responsável por administrar as máquinas que serão utilizadas como *Workers*;
- Os *Workers Nodes*, que são as máquinas que realmente executarão as tarefas que são enviadas pelo *Driver Program*. Se o Spark for executado de forma não-distribuída, a máquina desempenhará tanto o papel de *Driver Program* como de *Workers*.

Além da arquitetura, é importante conhecer os principais componentes do modelo de programação do Spark. Existem três conceitos fundamentais que serão utilizados em todas as aplicações desenvolvidas:

- *Resilient Distributed Datasets* (RDD): abstraem um conjunto de dados distribuídos no *cluster*, geralmente executados na memória principal. Estes podem estar armazenados em sistemas de arquivo tradicional, no *HadoopDistributed File System* (HDFS) e em alguns bancos de dados *Not Only SQL* (NoSQL), como Cassandra

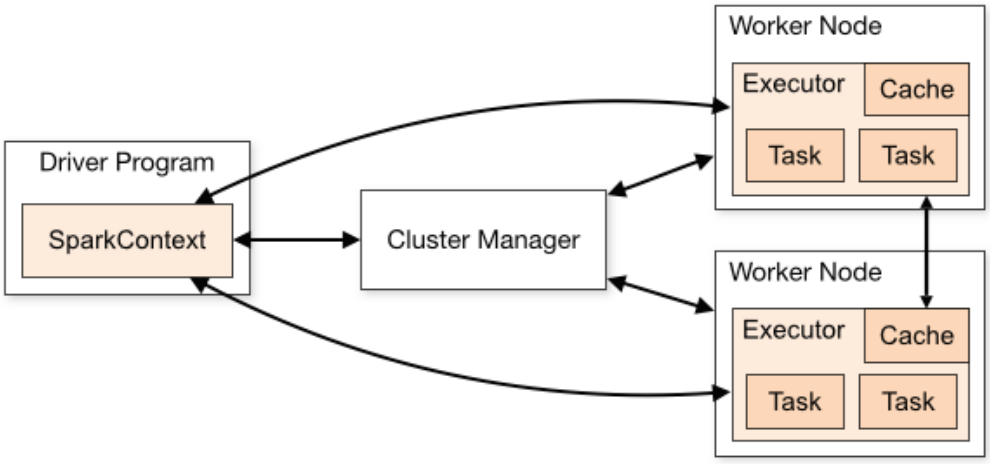


Figura 8 – Diagrama de funcionamento do Spark.
(Fonte: https://spark.apache.org/docs/latest/cluster-overview.html)

e HBase. Ele é o objeto principal do modelo de programação do Spark, pois são nesses objetos que serão executados os processamentos dos dados.

- Operações: representam transformações (como agrupamentos, filtros e mapeamentos entre os dados) ou ações (como contagens e persistências) que são realizados em um RDD. Um programa Spark normalmente é definido como uma sequência de transformações ou ações que são realizadas em um conjunto de dados.
- *Spark Context:* é o objeto que conecta o Spark ao programa que está sendo executado no ambiente. Ele pode ser acessado como uma variável de modo a parametrizar os recursos do Spark em um processamento.

3.2.2 Funcionamento do Spark

Conforme explicado por Islam et al. [2017], as aplicações são enviadas por meio do *Cluster Manager* para serem executados. Conforme verificado na Figura 7, o *Cluster Manager* do Spark pode ser executado em modo *cluster* em conjunto com o Apache Mesos [Hindman et al., 2011], o Hadoop YARN [Vavilapalli et al., 2013] ou a Amazon EC2 para execução de tarefas em ambientes distribuídos; e o modo *Standalone* para execuções em ambientes não-distribuídos.

O *Cluster Manager* suporta alocação estática e/ou dinâmica de recursos. Na

alocação de recursos estáticos, cada aplicação é executada com uma quantidade fixa de recursos que não podem ser alterados durante o ciclo de vida desta. No entanto, na alocação dinâmica de recursos, os recursos ociosos podem ser liberados para o *cluster* e qualquer outra aplicação em execução pode utilizá-los. Esses recursos também podem ser recuperados do *cluster* no futuro, se necessário.

O Spark usa RDD para manter os dados de uma maneira tolerante a falhas. Cada job/aplicação é dividido em vários conjuntos de tarefas chamados estágios que são interdependentes. Como característica de um *workflow* científico, todas essas etapas formam um GAD e cada estágio é executado um após o outro. A Figura 9 demonstra um mapeamento da execução de estágios e de tarefas executadas em cada estágio de uma aplicação no Spark visualizado através de um GAD.

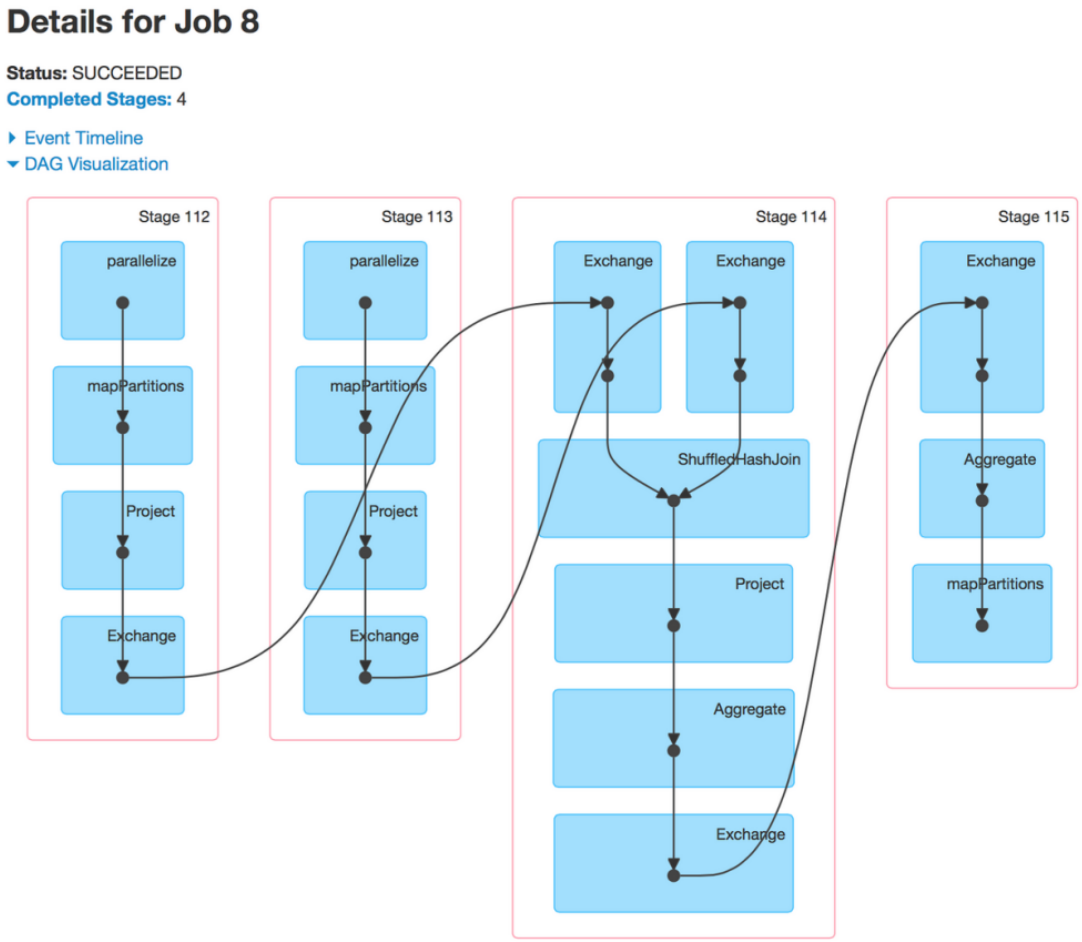


Figura 9 – Visualização em GAD referente a uma aplicação com vários estágios executados no Spark (Fonte: https://databricks.com)

3.3- Spark SQL

Embora a popularidade dos sistemas relacionais mostre que os usuários frequentemente preferem escrever consultas declarativas, a abordagem relacional é insuficiente para muitas aplicações de *big data*. O Spark SQL [Armbrust et al., 2015] é um módulo do Apache Spark que integra o processamento de dados de forma relacional com a API de programação funcional do Spark. Este permite que programadores do Spark aproveitem os benefícios do processamento de dados de forma relacional (por exemplo, consultas estruturadas e armazenamento otimizado), e permite a integração com outras bibliotecas do Spark (por exemplo, MLlib para aprendizado de máquina). Este fornece uma API para a estrutura de *Dataframes* do Spark que pode realizar operações relacionais em fontes de dados externas e em conjuntos de dados distribuídos no Spark.

O *Dataframes* é uma coleção distribuída de dados organizado em colunas rotuladas que fornece os benefícios dos RDDs com os benefícios do mecanismo de execução otimizado do Spark SQL. É conceitualmente equivalente a uma tabela em um banco de dados relacional ou um quadro de dados no R/Python. Isso fornece mais informações ao Spark sobre a estrutura de ambos os dados e a computação. Essas informações podem ser usadas para diversas otimizações. Esta ferramenta possui algumas propriedades importantes:

- Suporte ao processamento relacional tanto dentro dos programas Spark (em RDDs nativos) e em fontes de dados externas usando uma API amigável ao programador;
- Suporta facilmente novas fontes de dados, incluindo dados semiestruturados e bancos de dados externos passíveis de consultas federadas;
- Extensão a algoritmos analíticos avançados, como processamento de grafos e aprendizado de máquina.

Spark SQL usa um modelo de dados baseado em Apache Hive [Huai et al., 2014] para tabelas e *DataFrames*. Ele suporta todos os principais tipos de dados SQL, incluindo booleano, inteiro, duplo, decimal, string, data e timestamps, bem como tipos de dados complexos (ou seja, não atômicos): estruturas, matrizes, mapas de caracteres. Além disso, Spark SQL também suporta tipos e funções definidas pelo usuário (*User-Defined Functions* (UDF)).

Trabalhos recentes como o de Ferreira et al. [2018] e Chen et al. [2018] apontam bons resultados com o uso do Spark SQL na construção de *workflows* científicos, explorando a otimização de suas funções nativas.

3.4- Graphx e GraphFrames

A biblioteca GraphX foi projetada para permitir a construção de novas APIs para processamento de grafos de forma paralela e distribuída. Ao compor operações na interface *Resilient Distributed Graphs* (RDG), que é uma extensão do RDD do Spark, é possível expressar de forma compacta várias abstrações gráficas paralelas mais utilizadas.

Segundo Alemi et al. [2017], ela possui coleções de vértices e arestas que armazenam os seus dados de maneira distribuída. Armazena índices extras relacionados à estrutura do grafo de modo a acelerar o processo de união local e aumentar o desempenho da agregação. Para atingir esse objetivo, conforme demonstrado na Figura 10, cada aresta é agrupada pelo ID de seu vértice-fonte que armazena também o ID do vértice de destino. Ainda na Figura 10, cada vértice possui duas estruturas de dados: um *bitmask* para ativar a interseção de conjunto e uma tabela de roteamento que mapeia o conjunto de arestas para um determinado vértice.

Outra biblioteca alternativa para processamento de grafos no Spark é o GraphFrames, que pode combinar processamento relacional, correspondência de padrões e algoritmos de processamento de grafos de modo a otimizar os respectivos cálculos [Dave et al., 2016].

O GraphFrames suporta processamento de grafos em geral, semelhante ao GraphX. No entanto, os GraphFrames são criados utilizando a estrutura de *Dataframes* do Spark SQL, resultando em algumas vantagens importantes:

- API em Python, Java e Scala: o GraphFrames fornece API uniforme para todas as três linguagens de programação diferentemente do GraphX, onde todos os algoritmos estão disponíveis apenas na linguagem *Scala*;
- Consultas poderosas: o GraphFrames permite que os usuários formem consultas conhecidas e poderosas à API do Spark SQL;

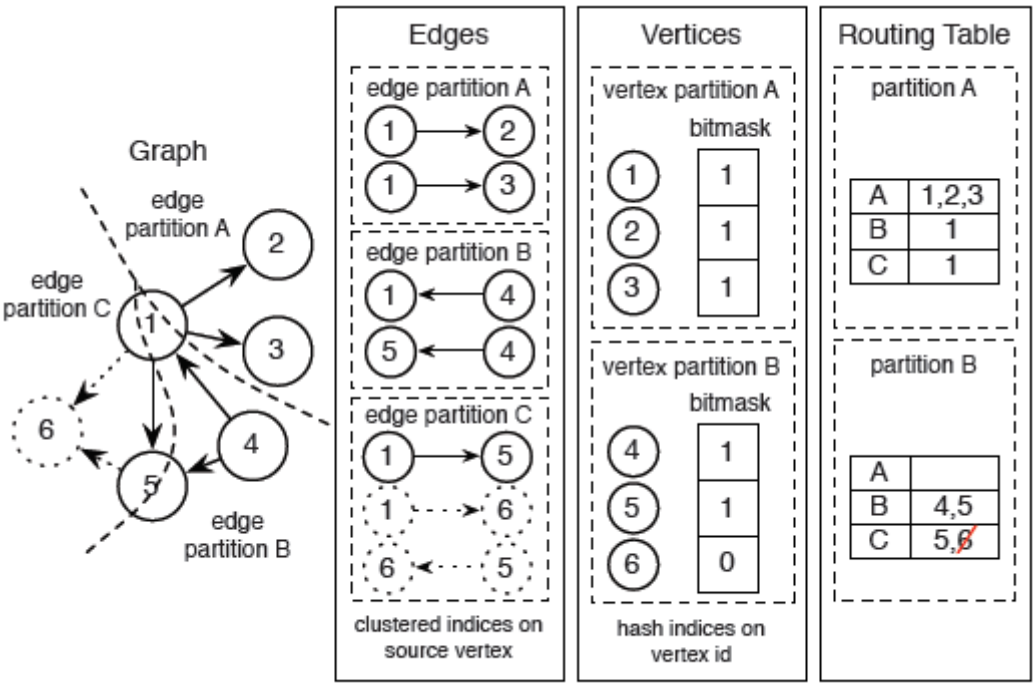


Figura 10 – Representação de um grafo na estrutura do GraphX [Xin et al., 2013]

- Manipulação de grafos: O GraphFrames suporta totalmente fontes de dados *DataFrame*, permitindo escrever e ler grafos usando vários formatos como JSON e CSV.

4- Metodologia

Neste capítulo descrevemos o desenho metodológico desenvolvido para atingir os objetivos da pesquisa. Na Seção 4.1, são descritos os passos seguidos para a composição bibliográfica desta pesquisa. Na Seção 4.2 é descrita a estrutura física, estrutural e operacional que dá sustentação ao *workflow* científico proposto. A Seção 4.3 apresenta as etapas de criação das bases de dados que alimentam as etapas de processamento. Na Seção 4.4 a estrutura de implementação computacional e as interações entre os diferentes ambientes são demonstradas com detalhes do que é executado em cada etapa do fluxo, apresentando as bibliotecas utilizadas nas linguagens Python e Java. Na Seção 4.5 descrevemos os testes de execução da ferramenta realizados para validação do portal científico utilizando como base resultados conhecidos na literatura de TEG. A Seção 4.6 apresenta a definição de cálculos de *speedup* e eficiência utilizados como métricas de avaliação de desempenho. E, por fim, a Seção 4.7 trata da disponibilização do Portal científico a pesquisadores para testes de novas e/ou existentes conjecturas de TEG. A Figura 11 fornece uma visão geral de todas as etapas metodológicas que são apresentadas de forma detalhada ao longo deste capítulo.

4.1- Busca bibliográfica

A composição bibliográfica desta pesquisa consistiu em buscas na base de periódicos da Capes por artigos que referenciassem os diferentes tópicos abordados nesta dissertação. Foi elaborado um mapa sistemático com buscas de conteúdos associados à pesquisa como definições e conceitos de TEG e exemplos de aplicações em Ciência de Computação onde uma simples busca com as palavras-chave *graphs-spectral-theory* retornou mais de 33 mil resultados. Utilizando a técnica de pesquisa *Snowballing*, verificamos que diversas produções possuem citações a trabalhos de Dragos Cvetković. Seu nome foi referenciado no filtro de pesquisa, de modo que obtivemos 473 artigos. Através de análise mais detalhada verificando o título e o *Abstract* de cada um, chegamos

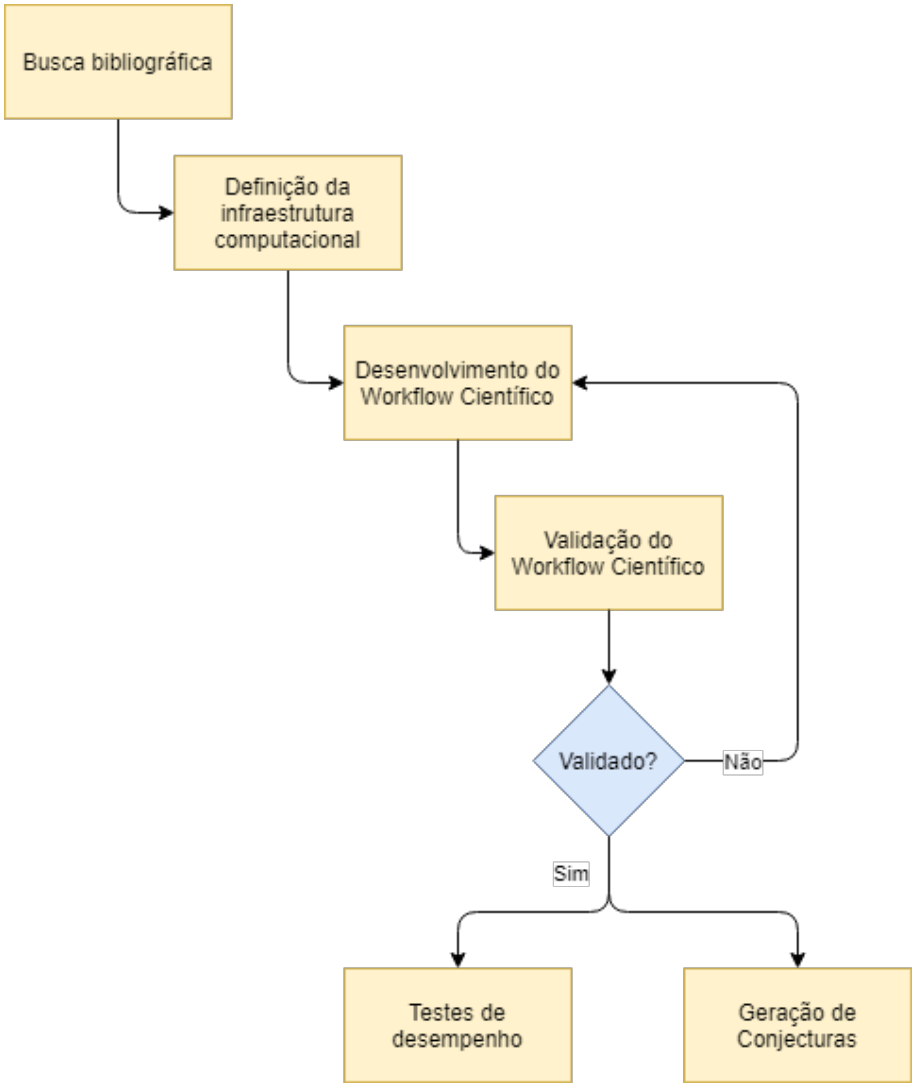


Figura 11 – Diagrama de etapas da metodologia de desenvolvimento.

a uma relação de 15 artigos importantes para leitura e que foram referenciados nesta dissertação.

Para verificação de *softwares* existentes que auxiliam pesquisadores em TEG, boa parte dos artigos analisados na pesquisa mencionada no parágrafo anterior utilizavam o AutoGraphix. De modo que foram feitas buscas no Google e GitHub por *softwares* e pacotes de códigos similares de apoio a pesquisas em TEG, onde o Graffiti, Nauty-traces, NewGraph, Mathchem, House of Graphs e Graph6Java foram também referenciados.

O próximo passo consistiu em elaborar um modelo de arquitetura de ambiente que deu sustentação aos objetivos desta pesquisa. Diante disto, foi elaborado um outro mapa sistemático com o objetivo de verificar exemplos de *workflows* científicos próprios para *Big Data* onde o processamento ocorre em ambiente paralelo e/ou distribuído e

com controle de falhas. Com o uso das palavras-chave *parallel-distributed-processing-system-scientific-workflow-big-data-fault-tolerant* identificamos 90 artigos com diferentes ferramentas, dentre as quais o Apache Spark mostrou boas referências em termos de consistência e desempenho.

4.2- Infraestrutura Física Computacional

O laboratório de computação da Escola de Informática & Computação (EIC) do CEFET/RJ conta com 13 servidores de alto desempenho para realização de pesquisas relacionadas a Ciência da Computação. Destes servidores, foi disponibilizado o servidor Antares para que as pesquisas fossem desenvolvidas. O servidor Antares possui dois processadores com 28 núcleos de processamento cada, 188 Gigabytes de memória RAM, espaço em disco de 1,8 Terabytes e com sistema operacional Linux Ubuntu 19.10. Este ambiente de uso do servidor é compartilhado com outros discentes. Com a finalidade de termos um ambiente de testes isolado e com as especificações desejadas, utilizamos todas as aplicações dentro do ambiente *Docker*. O *Docker* [Diedrich, 2015] é uma plataforma *Open Source* escrito em *Go*, que é uma linguagem de programação de alto desempenho desenvolvida pela *Google*, que facilita a criação e administração de ambientes isolados.

O *Docker* possibilita o empacotamento de uma aplicação ou ambiente inteiro dentro de um container, e a partir desse momento o ambiente inteiro torna-se portável para qualquer outro *Host* que contenha o *Docker* instalado. Dentro do desta estrutura foram criados quatro containers: um contendo o serviço *Tomcat* para o serviço WEB; um contendo o banco de dados *PostgreSQL* para servir como base de dados de apoio ao Portal para registro de contas de usuários, de submissões e resultados das pesquisas; um contendo o Spark Master que possui o *Driver Program* e o *Cluster Manager*, este utilizando o modo *Standalone* pois o ambiente não é distribuído; e outro contendo o Spark Worker, responsável pela execução das tarefas do *workflow* científico. Com essa estrutura, o Spark fica em um ambiente apropriado com todas as bibliotecas necessárias disponíveis e isolado de modo que qualquer atualização ou manutenção no *Host* não interfira na execução.

Após a criação da estrutura, o ambiente de acompanhamento de tarefas é exibido em um navegador conforme Figura 12. A Figura 13 demonstra a conexão entre os *containers* no ambiente *Docker*.

O código *Docker* do ambiente Spark utilizado nesta pesquisa foi obtido em `https://github.com/mvillarrealb/docker-spark-cluster`. Descreveremos em mais detalhes o *workflow* científico na seção seguinte.

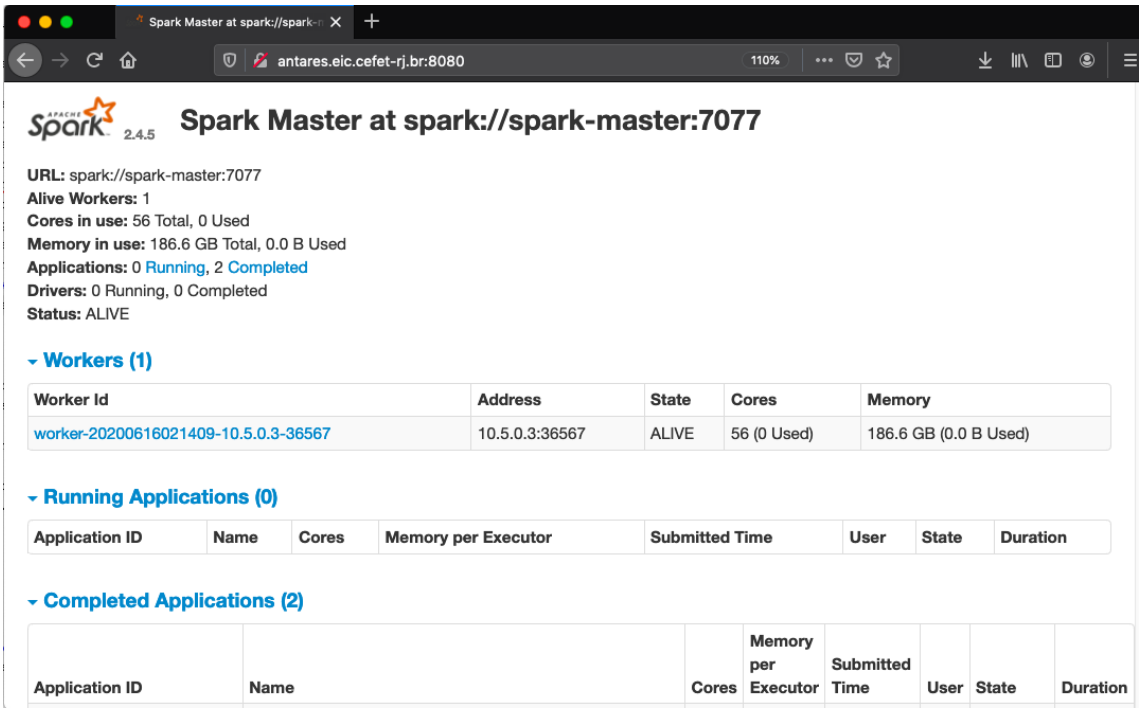


Figura 12 – Ambiente web para acompanhamento da execução de tarefas do Spark.

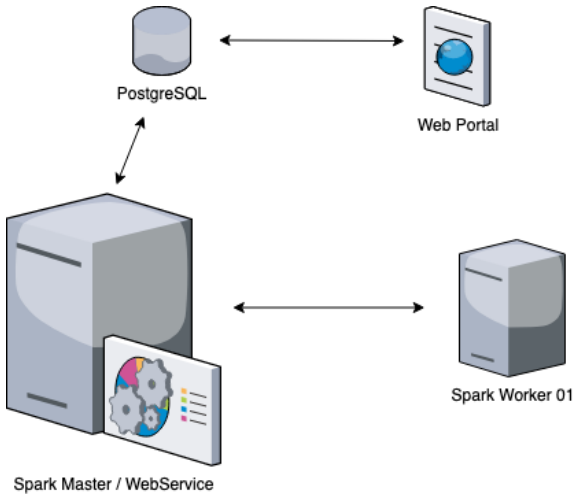


Figura 13 – Arquitetura do ambiente do Portal científico dentro do ambiente Docker.

4.3- Estrutura base e pré-processamento

O primeiro passo para a construção do *workflow* científico é a etapa de pré-processamento fazendo a composição da base de dados que alimentará o Spark. A base de dados contém o código do grafo em formato g6, a ordem n, os graus mínimo e máximo, a informação se o grafo é conexo, se é livre de triângulos, e se é bipartido. A geração dessa base foi realizada em duas etapas. Na primeira etapa todos os grafos simples com ordens 5 a 11 vértices foram gerados com o uso da rotina *geng* do Nauty-Traces e o tamanho total dessa base de dados possui aproximadamente 24 Gigabytes. Na segunda etapa de construção da base de dados desenvolvemos uma rotina em Python para classificar todos os grafos gerados na primeira etapa. Desse modo, para cada grafo foi associado uma *string* contendo: código g6 do grafo; os graus máximos e mínimos; 1 ou 0 se o grafo for conexo ou não; 1 ou 0 se o grafo for ou não livre de triângulos; 1 ou 0 se o grafo for bipartido ou não.

Uma vez construída a base de dados de grafos, para a etapa de processamento, o *workflow* científico demonstrado pela Figura 14 foi desenvolvido em duas linguagens diferentes. Inicialmente, desenvolvemos utilizando a linguagem Python uma vez que a ferramenta *SageMath* [Stein, 2007], disponível em `http://sagemath.org`, baseada nesta linguagem, já possui diversas bibliotecas prontas para manipulação de grafos. E posteriormente desenvolvemos um outro *workflow* em linguagem Java com auxílio das bibliotecas utilizadas no programa Graph6Java, disponível em `https://github.com/dragance106/graph6java` e outras auxiliares que serão descritas a seguir. Na próxima subseção será apresentada uma descrição geral de cada etapa do fluxo e como foi desenvolvido para cada linguagem.

4.4- Desenvolvimento do *workflow* científico

Esta seção apresenta a descrição de cada etapa do *workflow* científico a ser processado pelo Portal RioGraphX. Todas as etapas estão resumidas na Figura 14.

Figura 14 – Diagrama de etapas do *workflow* científico do Portal RioGraphX

Etapa A: nesta primeira etapa o usuário deve definir os parâmetros, que por sua vez definirão o problema de otimização a ser investigado. Os parâmetros a serem informados são os seguintes:

A1. Definição da função objetivo $f(.)$ que pode ter como domínio os seguintes invariantes de grafos: autovalores das matrizes A_G, L_G e Q_G, a saber λ_i, μ_i e q_i; ou os seus complementares, $\overline{\lambda_i}$, $\overline{\mu_i}$ e $\overline{q_i}$ respectivamente. A Tabela 3 lista os demais invariantes disponíveis nesta etapa de parametrização.

Invariantes disponíveis	Descrição
$[n_{min}, n_{máx}]$	intervalo das ordens n dos grafos que serão gerados
$[d_{g_{min}}, d_{g_{máx}}]$	intervalo dos graus mínimo e máximo dos grafos a serem gerados
k	número k de grafos a serem retornados na execução
ρ_i ou $\overline{\rho_i}$	autovalores da matriz de Distância D_G ou $\overline{D_G}$
γ_i ou $\overline{\gamma_i}$	autovalores da matriz de Distância Laplaciana QD_G ou $\overline{QD_G}$
α_i ou $\overline{\alpha_i}$	autovalores da matriz de Distância Laplaciana sem sinal SQD_G ou $\overline{SQD_G}$
β_i ou $\overline{\beta_i}$	autovalores da matriz ABC ABC_G ou $\overline{ABC_G}$
χ_G ou $\overline{\chi}_G$	número cromático de G ou de $\overline{G}$
ω_G ou $\overline{\omega}_G$	clique maximal de G ou de $\overline{G}$
d_G	valor máximo do grau dos grafos a serem gerados
m_G	valor máximo do número de vértices dos grafos a serem gerados
E	Energia da Matriz de Adjacência do grafo
LE	Energia da Matriz Laplaciana do grafo
SLE	Energia da Matriz Laplaciana sem sinal do grafo
E_D	Energia da Matriz de Distância do grafo
LE_D	Energia da Matriz de Distância Laplaciana do grafo
SLE_D	Energia da Matriz de Distância Laplaciana sem sinal do grafo
ABC ou $\overline{ABC}$	Índice de conectividade *Atom-Bond* de G ou de $\overline{G}$
ABC_{GG} ou $\overline{ABC_{GG}}$	Índice *Graovac-Ghorbani* de G ou de $\overline{G}$
Grafos livre de triângulos	Caso habilitado, apenas grafos livre de triângulos serão gerados
Grafos conexos	Caso habilitado, apenas grafos conexos serão gerados
Grafos bipartidos	Caso habilitado, apenas grafos bipartidos serão gerados

Tabela 3 – Relação dos parâmetros disponíveis para definição de função objetivo

A2. A função de otimização é definida como $f(x_1, x_2, \ldots, x_t)$ onde cada x_i, para $i = 1, 2, \ldots, t$, é um dos invariantes disponíveis na Tabela 3. Como exemplo, temos a função $f_G(\lambda_2, \chi, \overline{\lambda_2}) = \lambda_2 + \chi + \overline{\lambda_2}$. Demais sintaxes dos invariantes disponíveis estão apresentados no item 3 do Apêndice B.

A3. Tipos de otimização:

- *Maximize:* selecionar esta opção indica que a ferramenta irá determinar todos os k grafos que maximizam a função $f_G(x_1, x_2, \ldots, x_t)$.

- *Minimize:* similar à *maximize*, porém determina todos os k grafos que minimizam a função $f_G(x_1, x_2, \ldots, x_t)$.

Um exemplo de entrada da função objetivo é a função \mu_2 + \overline{\mu_2} que na conversão para expressão matemática é representada por $\mu_2+\overline{\mu_2}$ para visualização em tempo real pelo usuário. As restrições disponíveis são: exigir que o grafo seja conexo, ou livre de triângulo ou bipartido. No código Python, a função objetivo em formato latex é convertida para Unicode utilizando a biblioteca pylatexenc para ser processada nas etapas seguintes. No código Java, a conversão de latex para Unicode não ocorre de modo satisfatório e foram mantidas as nomenclaturas em latex para uso durante o processamento.

Etapa B: nesta etapa para cada $n \in [n_{min}, n_{máx}]$ é feita a geração massiva de todos os grafos simples de ordem n que atendam aos parâmetros estabelecidos na etapa anterior.

Para otimizar esta etapa, são carregados do banco de dados para a estrutura de *Dataframes* do Spark todos os grafos conexos simples de ordem n variando entre 5 e 11 vértices. A Tabela 4 fornece o número de grafos e grafos conexos que compõem a base de dados.

Ordem	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Todos os grafos	34	156	1.044	12.346	274.668	12.005.168	1.018.997.864
Grafos conexos	21	112	853	11.117	261.080	11.716.571	1.006.700.565
% de grafos conexos	61,8	71,8	81,7	90,0	95,0	97,6	98,8

Tabela 4 – Quantidade de grafos e grafos conexos com ordens de 5 a 11.

Com cada grafo armazenado seguem outras informações no registro na base de dados. Assim, os registros possuem as seguintes colunas:

- grafo: código do grafo em formato g6;
- ordem: ordem n do grafo;
- grau mínimo: grau mínimo do grafo;
- grau máximo: grau máximo do grafo;
- *triangle free*: valor 1 se o grafo é livre de triângulos, e 0 caso contrário;
- *connected*: valor 1 se o grafo é conexo, e 0 caso contrário;

- *bipartite*: valor 1 se o grafo é bipartido, e 0 caso contrário.

Esses dados são carregados para o fluxo de processamento do Spark através do módulo Spark SQL, módulo este que realiza consultas a dados estruturados dentro do ambiente Spark na forma relacional do SQL, através da chamada da função `spark.read().load().toDF()`. Somando todos os registros de grafos com as respectivas informações tem-se o valor 7.219.038.960 objetos no Spark, contabilizando cerca de 24 GigaBytes de dados.

Foram estudadas duas estruturas de dados para a geração dos grafos a serem manipulados, a saber: RDD e *Dataframes*, ambos apresentados no Capítulo 3. A biblioteca nativa do Spark provê APIs para carregamentos dos dados contidos na base de dados construída para as duas estruturas mencionadas acima nos códigos Python e Java. Elas foram utilizadas e testadas na construção desta etapa do fluxo, sendo a estrutura de *Dataframes* escolhida a ideal em ambos os códigos pela facilidade na manipulação, organização das informações e pela rapidez.

Etapa C: nesta, também de forma paralela entre os estágios do Spark, são feitos os cálculos dos invariantes presentes na função objetivo filtrados pelos devidos parâmetros informados pelo usuário para cálculos em ambiente Spark, com bibliotecas de apoio para cada código. De modo geral, o grafo em formato g6 é convertido para o formato de matriz, formada por um vetor de inteiros bidimensional (Int[][]). A partir desta matriz são calculados os seus autovalores e esses colocados em um vetor de números reais (Double[]), onde o primeiro elemento é composto pelo menor autovalor e o último pelo maior autovalor.

Ambas as estruturas de dados utilizam funções customizadas, UDFs do módulo SparkSQL, para cálculo dos invariantes no ambiente Spark com o auxílio de bibliotecas externas para cada código. No Python são utilizadas as bibliotecas NetworkX e Grinpy [Amos and Davila, 2017] para transformação dos grafos do formato g6 em matrizes para posterior cálculo dos invariantes e associá-los aos itens da função objetivo. No código Java foi utilizada a classe Graph do Graph6Java para execução da mesma tarefa.

Etapa D: Com todos os grafos carregados e os seus respectivos invariantes calculados, nesta é calculada e avaliada a função objetivo para cada objeto de grafo $G \in G_n$ obtido após o pré-processamento e filtragem na Etapa **B**, também através de UDF criada para este fim e com bibliotecas de apoio para cada código. De modo que é realizada

a substituição das variáveis da string do formato latex para expressões matemáticas possibilitando a conversão de cada elemento da string para os valores de cada invariante calculado na etapa anterior. Isto é, seja uma expressão $A + B$ em formato string, os parâmetros A e B são substituídos pelos respectivos valores de invariantes do grafo que está sendo analisado em formato Double. A função Python utilizada nesta etapa é a eval(), nativa da linguagem. Em Java, após verificação de diversas bibliotecas, como MathEval e EXPR, a biblioteca Parsii (disponível em `https://github.com/scireum/parsii`) obteve a melhor eficiência no processamento da função objetivo. Os resultados dos cálculos desta etapa são incorporados à estrutura de dados do Spark através da função `WithColumn()` da estrutura de *Dataframes*, ocorrendo a criação de uma coluna com esses valores para cada linha de grafo.

Etapa E: esta etapa é responsável pela ordenação do resultado da função de acordo com o tipo informado na Etapa **A** (*maximize* ou *minimize*). Esta etapa é a mais crítica do *workflow*, pois nela os dados contidos na estrutura de dados do Spark são ordenados utilizando o método `OrderBy()` no *Dataframes*, nativo do Spark. Caso o usuário na etapa de parametrização deseje minimizar a função objetivo, os dados são ordenados em ordem crescente utilizando o parâmetro `asc()`, caso deseje maximizar, os dados são ordenados de forma decrescente, utilizando o parâmetro `desc()`, para que os resultados desejados fiquem no topo da estrutura de dados. Através de pesquisas na documentação do Spark, verificamos que este algoritmo possui complexidade $O(N\ logN)$, onde N representa quantidades de grafos carregados para execução parametrizados na etapa A. Este possui dois passos: o primeiro passo é distribuir os dados uniformemente entre as partições paralelas para, no segundo passo, realizar um *merge-sort* distribuído onde todas as partições paralelas são ordenadas para em seguida ser realizada a mesclagem entre elas.

Etapa F: após obter os dados ordenados, nesta etapa os *k* melhores grafos que satisfazem a função objetivo são agrupados pela ordem n do grafo utilizando as funções `filter(`n`)` e `limit(`k`)`, também nativas do Spark.

Etapa G: Nesta última etapa, é feita a geração das figuras dos grafos para posteriormente serem incorporadas ao relatório em formato PDF contendo as informações de submissão do usuário e os resultados do processamento do *workflow*. Após gerado

o relatório, o mesmo fica disponível para *download* pelo usuário e uma mensagem é disparada para o endereço eletrônico do usuário informando que o mesmo encontra-se disponível. No código Python, os dados são coletados para um vetor da biblioteca NumPy [Walt et al., 2011] usando a função *collect()*, também nativa do Spark. Este vetor é utilizado para geração das figuras dos grafos através da biblioteca matplotlib [Hunter, 2007] que posteriormente são utilizados para criação do arquivo PDF dos resultados em conjunto com a biblioteca fpdf [Berlitz, 2007]. No código Java, os dados são coletados utilizando a função *collectAsList()* para um objeto List<Row>. Cada grafo contido neste objeto é convertido para extensão .dot para posteriormente serem geradas as figuras através da biblioteca GraphViz [Ellson et al., 2001]. Por fim, o relatório PDF é confeccionado através da biblioteca itextpdf [Bruno, 2009].Um modelo de relatório gerado encontra-se no Apêndice A desta pesquisa.

A Tabela 5 demonstra a visão geral do *workflow* científico construído contendo as atividades que são executadas em ambiente paralelo e não-paralelo, bem como as entradas e saídas de dados para cada código desenvolvido. Os códigos-fonte das duas linguagens desenvolvidas para o *workflow* estão disponíveis em `https://github.com/danielfeol/`.

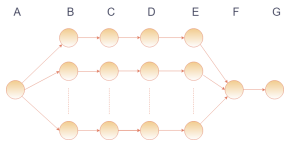
Visão conceitual do workflow	Atividade	Ambiente	Biblioteca	Entrada	Saída
	Etapa A: Parametrização	WEB	JSP	1.Função objetivo 2.Intervalo de ordem 3.Intervalo de grau 4.Se somente conexos 5.Se somente bipartidos 6.Se somente livre de triângulos	Escrita em banco de dados
	Etapa B: Carregamento dos dados de grafos	SPARK	Conector Postgres	Leitura de banco de dados com aplicação de filtros da parametrização	Construção do Dataframe (Figura X1)
	Etapa C: Cálculo dos invariantes	SPARK	Python: NetworkX e Grinpy Java: Graph6java	Invariante em formato latex	Valor calculado do invariante
	Etapa D: Cálculo da função objetivo	SPARK	Python: eval() Java: Parsii	Substituição da string latex pelos respectivos valores	Resultado da função objetivo e incorporação ao Dataframe (Figura X2)
	Etapa E: Ordenação	SPARK	Spark: OrderBy()	Dataframe	Dataframe ordenado (Figura X3)
	Etapa F: Selecionar k grafos	SPARK	Python: Numpy array Java: List<row>array	Dataframe ordenado	Vetor de objetos ordenado
	Etapa G: Geração de imagens e relatório	SPARK	Python: matplotlib e fpdf Java: GraphViz e Itextpdf	Vetor de objetos ordenados	Relatório final

Tabela 5 – Visão geral das etapas do *workflow* científico

4.5- Validação do *workflow* científico

Finalizada a criação dos dois códigos do *workflow*, este passo da metodologia consistiu na validação das informações produzidas ao fim da execução de cada código. Verificamos na comunidade científica trabalhos com conjecturas recentes para realização de testes. Um exemplo para testes considerou a desigualdade $\mu_2(G) + \mu_2(\overline{G}) \leq 2n - 2$, em que $\overline{G}$ é o grafo complementar de G. Para testar se essa desigualdade é verdadeira, usamos a função $f(\mu_2, \overline{\mu_2}, n) = n-2-(\mu_2(G)+\mu_2(\overline{G}))$ e exigimos apenas grafos conexos. Após a minimização, se o RioGraphX retornar um grafo como $f(\mu_2, \overline{\mu_2}, n) \leq 0$, teremos um contraexemplo e, portanto, a conjectura será refutada. Por outro lado, se $f(\mu_2, \overline{\mu_2}, n) \geq 0$ para todos os grafos, temos uma forte indicação de que a conjectura pode ser verdadeira. Além disso, podemos selecionar os grafos em que $f(\mu_2, \overline{\mu_2}, n) = 0$ e estender a conjectura apresentando os grafos extremais. Depois de executar o RioGraphX para todos os grafos de ordem $5 \leq n \leq 11$, nenhum contraexemplo foi encontrado e os grafos extremais obtidos motivaram a declaração da Conjectura 5 em Grijó et al. [2019]. Outros invariantes como os utilizados na conjectura de Nordhaus–Gaddum [Huang and Lin, 2019] e [Levene et al., 2019], o índice Graovac-Ghorbani [Pacheco et al., 2020] e o índice de conectividade Atom-Bond [Škrekovski et al., 2019] foram calculados.

Para apoio à verificação da consistências dos cálculos obtidos, acessamos o ambiente CoCalc [SageMath, 2019], um Portal que possui ferramentas para cálculos matemáticos em diversas linguagens diferentes disponível em `https://cocalc.com/` e reproduzimos os testes realizados. Em todos os casos, todos os valores e as respectivas figuras dos grafos verificados estiveram em conformidade com os resultados obtidos.

4.6- Testes de desempenho

A realização de testes com o portal científico diz respeito principalmente à questão do tempo de resolução dos problemas dados como entrada. Naturalmente, o tempo de execução cresce exponencialmente conforme a ordem n do grafo é incrementada, conforme quantitativo de grafos informados na Tabela 4. Dessa forma, com a finalidade

de verificar o ganho de desempenho com o uso do processamento paralelo do Spark realizamos exaustivos testes de execução de modo a aferir as médias de tempos. Com base nisso calculamos duas métricas de desempenho: *speedup* e eficiência.

Gustafson [1988] definiu *speedup* como a proporção do tempo de execução serial do melhor algoritmo sequencial para resolver um problema e o tempo gasto pelo algoritmo paralelo para resolver o mesmo problema nos processadores. Ele é calculado com a seguinte fórmula:

$$S = \frac{T_s}{T_p},$$

onde T_s é o tempo de execução da atividade sem paralelismo e T_p é o tempo de execução da atividade com paralelismo.

A *eficiência* é definida como a proporção de aceleração em relação ao número de processadores. A eficiência mede a fração de tempo durante a qual um processador é útil e é obtida de acordo com a seguinte fórmula:

$$E = \frac{S}{p} = \frac{T_s}{pT_p},$$

onde p é o número de processadores utilizados em atividade paralela. Todo o detalhamento dos testes realizados e análise do resultados estão no Capítulo 6.

4.7- Geração de Conjecturas em Teoria Espectral de Grafos

O portal foi disponibilizado nos meses de Outubro e Novembro de 2020 para alunos de mestrado e doutorado utilizarem a ferramenta em seus problemas de pesquisa no sentido de explorações de problemas de TEG em aberto e a geração de novas conjecturas. Apresentamos abaixo o exemplo cedido pelo doutorando Diego Júlio Pacheco, aluno do Programa de Pós-Graduação em Engenharia de Produção do CEFET-RJ.

DEFINIÇÃO DO PROBLEMA: sejam ABC_{GG} e $\overline{ABC}_{GG}$ os índices de Graovac-Ghorbani do grafo G e do seu complementar, respectivamente. Deseja-se determinar uma cota inferior, l_1, e os grafos extremais da expressão (1):

$$l_1 \leq ABC_{GG} + \overline{ABC}_{GG}. \quad (1)$$

A desigualdade apresentada em (1) é conhecida na literatura como desigualdade do tipo Nordhaus-Gaddum. Para a exploração deste problema foram feitas submissões com as seguintes características:

- Grafos com ordem variando em $5 \leq n \leq 10$;
- Restrição de conexidade: alguns experimentos exigiram conexidade do grafo e outros não.
- Função objetivo: $f(G) = ABC_{GG} + \overline{ABC}_{GG}$;
- Opção de otimização: Minimizar;
- Restrição de graus: 0 a $n-1$.

O tempo de resposta do portal para os experimentos com $n = 10$ foram de, em média, 1 minuto. A partir dos experimentos, a Conjectura 4.1 foi elaborada com o auxílio do portal RioGraphX.

Conjectura 4.1. *Seja G um grafo conexo de ordem $n \geq 5$ e $\overline{G}$ o seu complementar. Então,*

$$2\sqrt{\frac{n-2}{n-1}} + (n-2)\frac{\sqrt{2}}{2} + (n-3)\left(\sqrt{\frac{n-2}{3(n-3)}} + \sqrt{\frac{2}{3}}\right) \leq ABC_{GG}(G) + ABC_{GG}(\bar{G}).$$

A igualdade é válida se e somente se $K_{2,n-2} - e$.

5- O portal científico

Neste capítulo é apresentado com maiores detalhes o portal científico implementado. Este portal é composto por uma interface WEB no *front-end* e pelo *workflow* científico, descrito no Capítulo 4, no *back-end*.

O *front-end* do Portal consiste em um ambiente WEB desenvolvido em linguagem Java onde usuários criam contas para acesso às funcionalidades de pesquisa em TEG. No primeiro acesso, o usuário deverá criar sua conta clicando no *link "Create your account"*. Neste formulário ele deverá informar seu nome, endereço eletrônico e senha de no mínimo oito caracteres. Após o preenchimento basta clicar no botão *"Create account"*, conforme Figura 15. Com a conta criada, o próximo passo é ativá-la. Para isto o usuário receberá no endereço eletrônico recém-cadastrado um *link* de ativação onde ele deverá clicar para ativar a conta. Após a ativação, o usuário estará apto a fazer o login no Portal e acessar o ambiente de submissão de pesquisas.

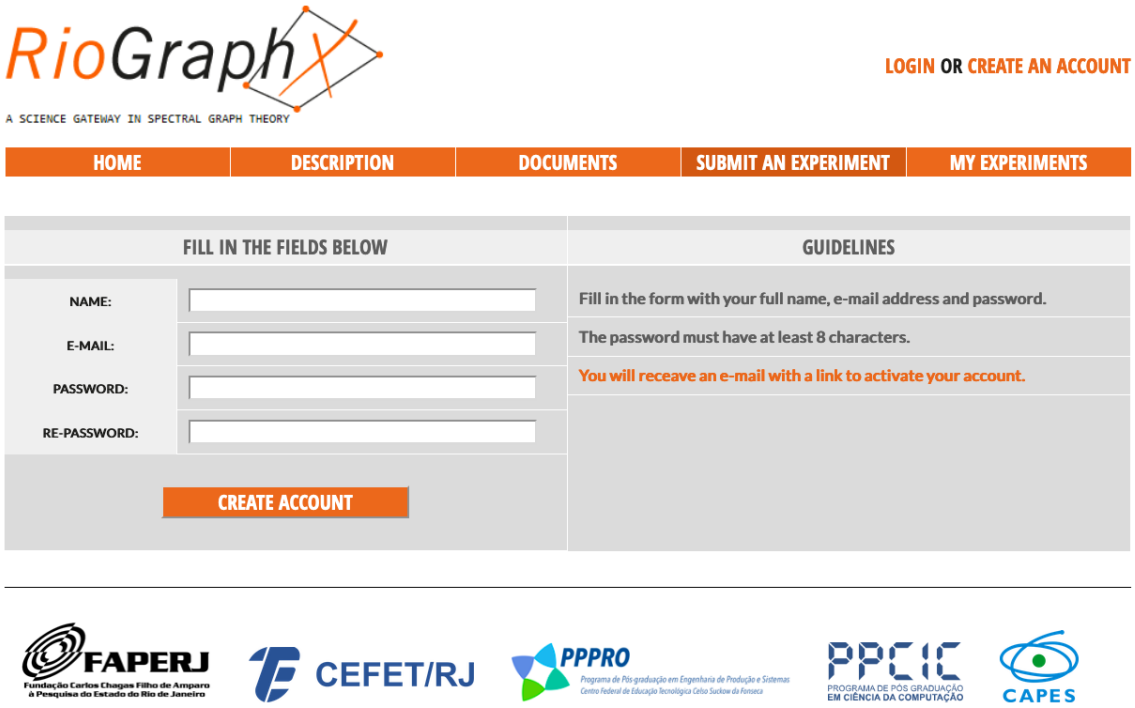


Figura 15 – Formulário para criação de conta no Portal científico.

Uma vez autenticado, o usuário tem disponível um formulário dinâmico e interativo

(demonstrado pela Figura 16) no qual preenche os seguintes campos para submissão de sua pesquisa:

- **Optimization function e Min/Max**. É a função de otimização combinatória na qual o usuário deseja verificar que grafos a minimizam ou a maximizam, com base nos invariantes informados como parâmetro na função. O preenchimento deve ser feito em codificação *latex* e, conforme o usuário preenche o campo, é exibida na tela a conversão da expressão da função do formato latex para o formato de expressão matemática;
- **Minimum/Maximum order**, critério de pesquisa para determinar o intervalo das ordens $[n_{min}, n_{máx}]$ dos grafos a serem processados;
- **Minimum/Maximum degree**, critério de pesquisa para determinar os graus do intervalo $[d_{g_{min}}, d_{g_{máx}}]$ dos grafos a serem processados;
- **Number of results to show**, número k de grafos por ordem n a serem processados.
- **Only generate triangle free graphs?**, parâmetro que define que apenas grafos livre de triângulos devem ser processados.
- **Only generate connected graphs?**, parâmetro que define que apenas grafos conexos devem ser processados.
- **Only generate bipartide graphs?**, parâmetro que define que apenas grafos bipartidos devem ser processados.

A Figura 16 demonstra um exemplo de submissão de trabalho ao Portal, onde o usuário deseja verificar os grafos extremais que minimizam a função $\mu_2 + \overline{\mu}_2$ (soma do 2º maior autovalor da Matriz laplaciana de G com o 2º maior autovalor da Matriz laplaciana do grafo $\overline{G}$), e, de acordo com o preenchimento dos campos abaixo da função, os resultados esperados neste exemplo são no máximo dez grafos por ordem (campo *Number of results to show*) com valores de ordem de 5 a 10 (campos *minimum/maximum order* respectivamente), valores de graus dos vértices entre 0 e 10 (campos *minimum/maximum degree* respectivamente). Ao passo que o usuário preenche o campo *Optimization function* em codificação latex, acima do formulário de submissão é exibida, em tempo real, a função convertida do código latex para expressão matemática. À direita do formulário constam os tempos médios de execução do *workflow* para cada ordem do

grafo. Ao término do preenchimento, o usuário deverá clicar no botão *"Submit Job"*. Caso algum parâmetro do formulário não esteja preenchido, o sistema fará a crítica solicitando o correto preenchimento. Caso a submissão bem seja sucedida, o usuário será automaticamente redirecionado para a página *"My Experiments"* para acompanhar o processamento da submissão.

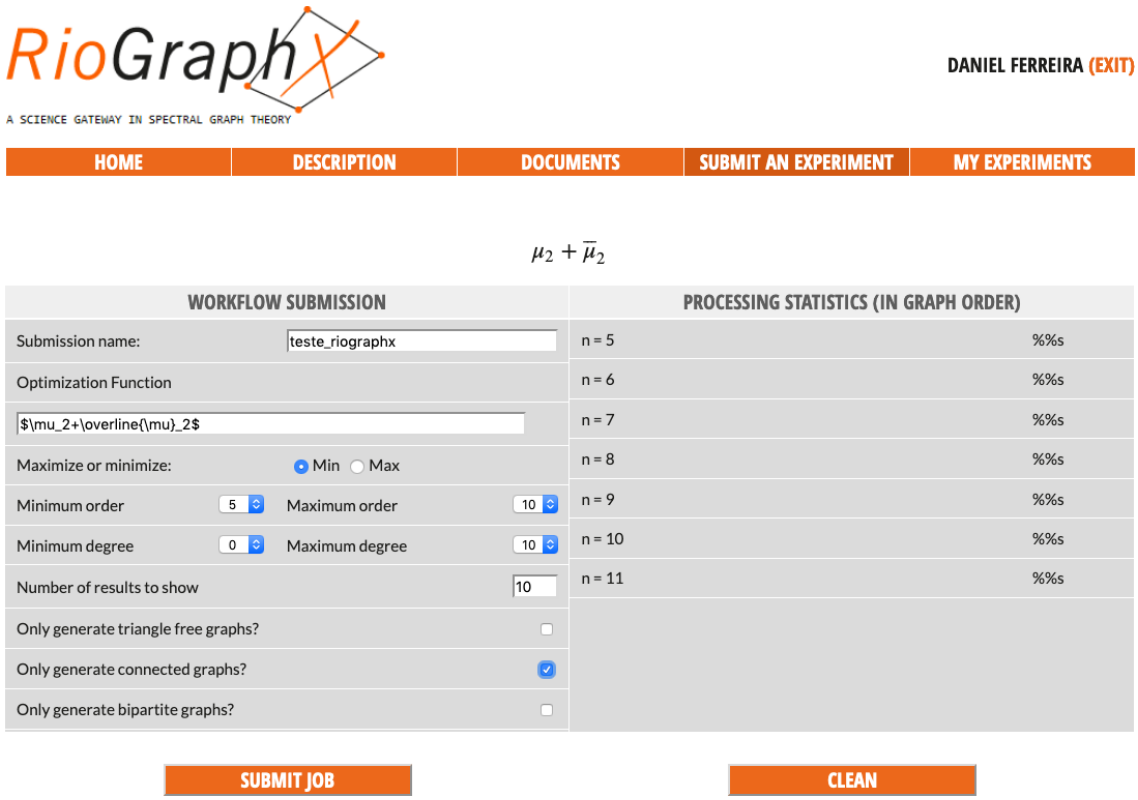


Figura 16 – Formulário dinâmico para submissão de trabalho.

O usuário pode acompanhar em sua conta todas as requisições já efetuadas com os respectivos *status* de andamento: *PROCESSING QUEUE* (em fila de processamento), *PROCESSING* (em processamento) ou *FINISHED* (finalizada) e o tempo de execução registrado de cada requisição, conforme observado na Figura 17. Caso ocorra algum problema durante o processamento, será exibido o *status ERROR* e nenhum relatório estará disponível. Que pode ter sido ocasionado por imputação de invariantes inexistentes no portal ou por algum erro de sintaxe na função objetivo.

Ao fim do processamento de uma requisição, estará disponível um arquivo em formato PDF, através do *link download*, contendo o relatório de informações dos grafos obtidos dentro dos parâmetros informados no formulário de submissão e as respectivas imagens. A listagem de resultados de grafos seguirão ordem de classificação definido no

formulário. Uma mensagem eletrônica é encaminhada ao usuário para notificar que sua requisição foi concluída.

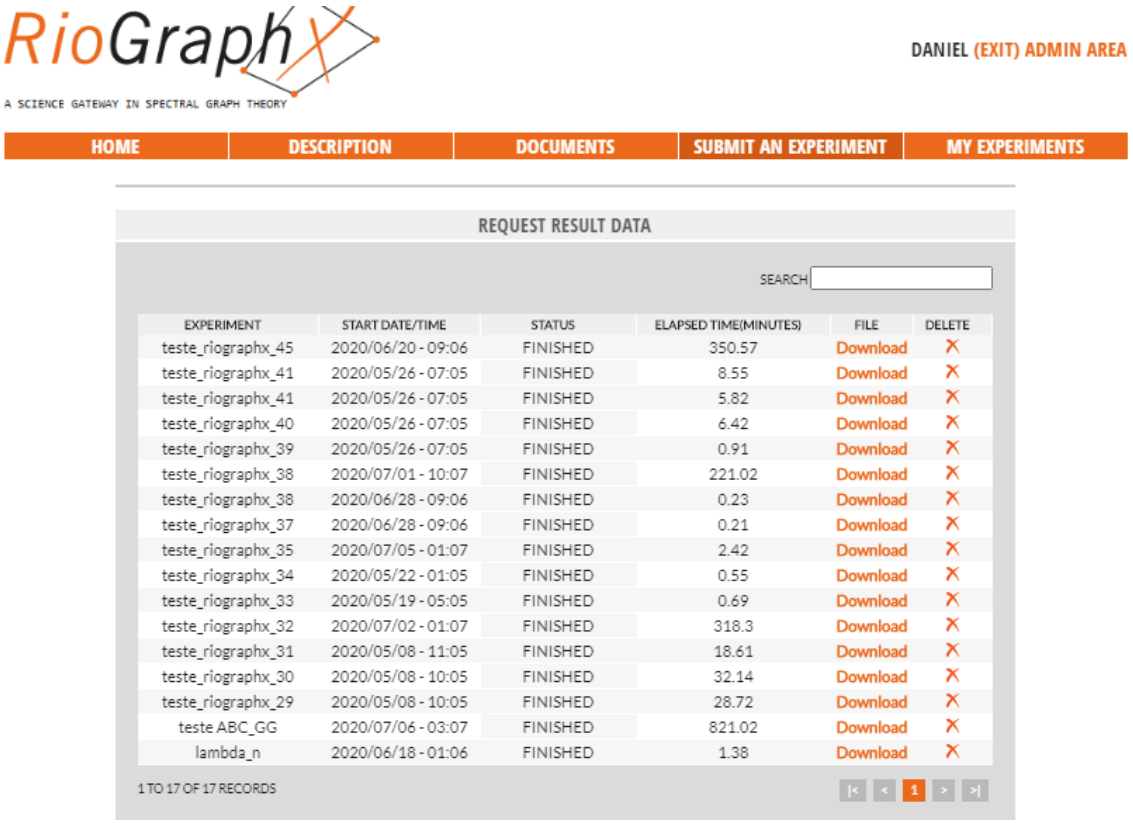


Figura 17 – Página contendo todos os experimentos realizados pelo usuário

Esta ferramenta possui proposta similar ao sistema AGX, ou seja, achar grafos extremais que minimizam ou maximizam a função combinatória com um ou mais invariantes como parâmetros. O diferencial desta ferramenta proposta são os seguintes:

- Sistema em ambiente WEB que dispensa uso de recursos processamento na máquina local do usuário;
- Fornece uma interface *on-line*, na qual os usuários podem formular conjecturas matemáticas e testá-las sem codificar rotinas computacionais.
- Todos os usuários terão conta de acesso ao Portal de modo que este possa acompanhar os resultados de suas submissões. Ou seja, o usuário poderá submeter várias requisições de cálculo e estas estarão listadas em sua conta. De modo que o usuário não precisará aguardar o fim do processamento de uma requisição para submeter outra;

- Notificação por mensagem eletrônica quando alguma submissão é concluída;
- Relatório detalhado dos cálculos realizados com as respectivas imagens dos grafos encontrados pela ferramenta.

6- Testes de desempenho e Avaliação dos Resultados

Este capítulo apresenta os testes de desempenho realizados no Portal científico e avaliação dos mesmos. Na Seção 6.1 é demonstrado um comparativo entre as médias de tempos de execução obtidas com os códigos desenvolvidos em Java e Python. Com o melhor algoritmo verificado, na Seção 6.2 são feitos os cálculos de *speedup* e Eficiência com diferentes configurações do ambiente Spark. Por último, na Seção 6.3, é feita a avaliação de todo o *workflow* cientifico baseado nos resultados dos testes apresentados nas seções anteriores.

6.1- Comparativo Java vs Python

Com os dois códigos do *workflow* científico desenvolvidos nas linguagens Java e Python e validados com base nas verificações descritas na Seção 4.5, realizamos baterias de testes divididos em etapas que são definidas pela ordem do grafo com o intuito de verificar qual deles é o mais rápido. Cada etapa consistiu em 10 execuções consecutivas do *workflow* utilizando a capacidade máxima do ambiente Spark, ou seja, os 56 núcleos de processamento disponíveis, e ao final de cada foram calculados os tempos médios e desvios-padrão. Os parâmetros de entrada dos testes consistiram em utilizar uma função objetivo simples como $\mu_2 + \overline{\mu_2}$, utilizada também nos testes de validação. Utilizamos o maior intervalo de graus mínimo/máximo possíveis (a saber, grau 0 e $n-1$) e não restringimos a busca para grafos conexos, nem livre de triângulos e nem bipartidos para que o máximo de grafos disponíveis na base de dados fossem processados. Foram realizados testes para grafos de ordem $n = 5, 6, 7, 8, 9$ e 10 e para todos os grafos do intervalo de ordem $5 \leq n \leq 10$ de uma vez só. A Tabela 6 apresenta os tempos médios e desvios-padrão, em segundos, para cada etapa da bateria de testes mencionada.

Observe na Tabela 6 que a implementação do workflow em Python obteve tempos médios melhores para grafos pequenos com número de vértices de 5 a 7. Já para os grafos com mais de 7 vértices, o código desenvolvido em Java obteve médias de tempos

Ordem do grafo	Média de tempo obtida no Python	Média de tempo obtido no Java
$n = 5$	$6,922 \pm 0,190$	$11,320 \pm 1,087$
$n = 6$	$7,091 \pm 0,075$	$10,795 \pm 0,165$
$n = 7$	$8,854 \pm 0,080$	$10,895 \pm 0,254$
$n = 8$	$32,413 \pm 0,309$	$14,677 \pm 0,841$
$n = 9$	$338,418 \pm 2,522$	$56,784 \pm 2,050$
$n = 10$	$1.587,960 \pm 50,519$	$288,324 \pm 157,650$
$5 \leq n \leq 10$	$1.596,350 \pm 182,855$	$381,343 \pm 139,421$

Tabela 6 – Média de tempo de execução do *workflow*

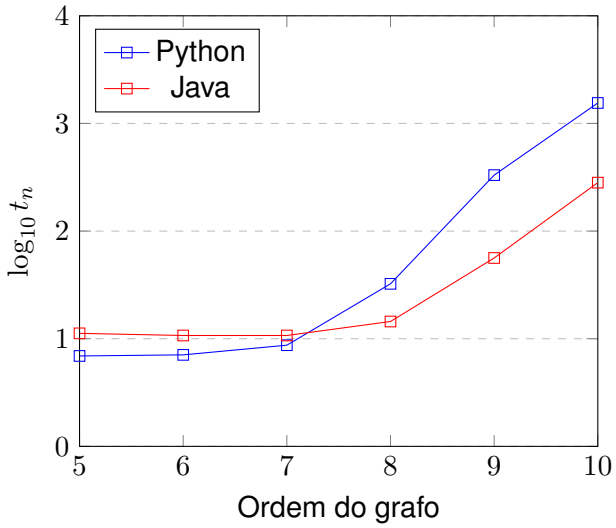


Figura 18 – Tempos médios de execução (t_n) para cada ordem n

de execução até 5x mais rápida que o código em Python. Esse comportamento pode ser observado pelo Gráfico 18 onde o eixo x representa a ordem n do grafo e o eixo y são os valores de tempos médios (t_n) apurados em uma escala logarítmica $y = \log_{10} t_n$ para cada ordem. Com isso, a implementação do *workflow* em Java mostrou-se mais vantajoso para grafos maiores tendo em vista que Java é uma linguagem compilada e Python é interpretada, dinamicamente tipada, Python tende a ser mais lenta para lidar com *Big Data*. As tabelas contendo todos os tempos de execuções registrados nesta Seção estão no Apêndice C desta dissertação.

Na próxima seção apresentamos os cálculos para medir o *speedup* e a eficiência do algoritmo em Java no contexto de incremento do número de processadores.

6.2- Cálculos de *speedup* e eficiência

Para esta pesquisa, utilizaremos a Lei de Amdahl [Gustafson, 1988] como modelo de desempenho de um código paralelizado. Este modelo fornece o *speedup* (tempo de execução da versão serial dividido pelo tempo de execução da versão paralela) potencial máximo que um programa pode ter, definido em função da fração do código que pode ser paralelizada.

Para aplicar a Lei de Amdahl, será feita a razão do tempo total de execução serial (T_s) do *workflow* científico utilizando apenas um núcleo de processamento ($p = 1$) pelo tempo total de execução paralela (T_p), utilizando metade ($p = 28$) e todos os núcleos ($p = 56$) do servidor utilizado para testes. A Tabela 7 apresenta as médias de tempos de execução do código desenvolvido em Java utilizando apenas um núcleo, 28 núcleos e 56 núcleos, com os respectivos valores de *speedup*. Nesta etapa, acrescentamos aos testes todos os grafos de ordem $n = 11$.

Ordem do grafo	**linear (1 núcleo)**	**Paralela (28 núcleos)**		**Paralela (56 núcleos)**	
Ordem	Média(s)	Média(s)	*speedup*	Média(s)	*speedup*
$n = 5$	$14,039 \pm 4,593$	$9,650 \pm 0,382$	$1,455$	11,320 $\pm 1,087$	$1,240$
$n = 6$	$10,513 \pm 0,749$	$9,875 \pm 0,201$	$1,065$	$10,795 \pm 0,165$	$0,974$
$n = 7$	$10,062 \pm 0,555$	$9,772 \pm 0,293$	$1,030$	$10,895 \pm 0,254$	$0,924$
$n = 8$	$16,524 \pm 1,305$	$13,321 \pm 0,712$	$1,240$	$14,677 \pm 0,841$	$1,126$
$n = 9$	$96,355 \pm 16,081$	$52,901 \pm 2,769$	$1,821$	$56,784 \pm 2,050$	$1,697$
$n = 10$	$2.483,324 \pm 17,464$	$215,513 \pm 60,511$	$11,523$	$288,324 \pm 157,650$	$8,613$
$n = 11$	$125.070,806 \pm 3431,886$	$55.381,157 \pm 60,511$	$8,465$	$12.708,400 \pm 918,807$	$14,775$
$\leq n \leq 10$	$2.476,998 \pm 71,770$	$342,027 \pm 93,066$	$7,242$	$381,343 \pm 139,421$	$6,495$
$n = 10(*)$	$2.483,324 \pm 17,464$	$131,363 \pm 3,328$	$18,904$	$116,911 \pm 4,557$	$21,241$
$5 \leq n \leq 10(*)$	$2.476,998 \pm 71,770$	$135,398 \pm 3,805$	$18,294$	$112,441 \pm 4,697$	$22,029$

Tabela 7 – Médias de tempos e *speedup* do algoritmo desenvolvido em Java

Os testes foram executados de forma análoga aos realizados na seção 6.1, ou seja, baterias de 10 execuções, para cada ordem n, onde uma iteração era iniciada exatamente ao término de outra. Foi utilizada a mesma função objetivo $\mu_2 + \overline{\mu_2}$ dos testes anteriores com o parâmetro de minimização dos resultados da mesma.

Para os testes realizados para grafos com ordem $n \geq 10$, foi observado que os tempos de execuções sofriam acréscimos a cada iteração, como pode ser observado pelos altos valores de desvio-padrão na Tabela 7. Investigando a causa, foi verificado um problema de superaquecimento dos processadores do servidor Antares por conta das execuções consecutivas. De modo que outro teste similar ao anterior foi elaborado com

intervalos de cinco minutos entre cada iteração, apenas para execuções que envolviam grafos de ordem $n = 10$ e $n = 11$ utilizado 28 e 56 núcleos de processamento em cada bateria de testes. Os tempos foram marcados com asterisco (*) na Tabela 7 e nos Gráficos 19 e 20, diante disto, pode-se verificar uma grande melhora nos tempos médios de execução e desvios-padrões mais uniformes. Nos testes com grafos de ordem $n = 11$ não houve diferença nos tempos com e sem o referido intervalo. Os valores dos testes utilizando apenas 1 núcleo não foram repetidos pois não foi observada ocorrência de superaquecimento nestes. As tabelas contendo todos os tempos de execuções registrados nesta Seção estão no Apêndice C desta dissertação.

O Gráfico 19 apresenta os valores de *speedup* calculados. O eixo vertical representa a ordem n dos grafos envolvidos na instância de testes e o eixo horizontal os valores absolutos calculados de *speedup*. A barra azul representa o *speedup* dos testes utilizando 28 núcleos e a barra laranja, 56 núcleos.

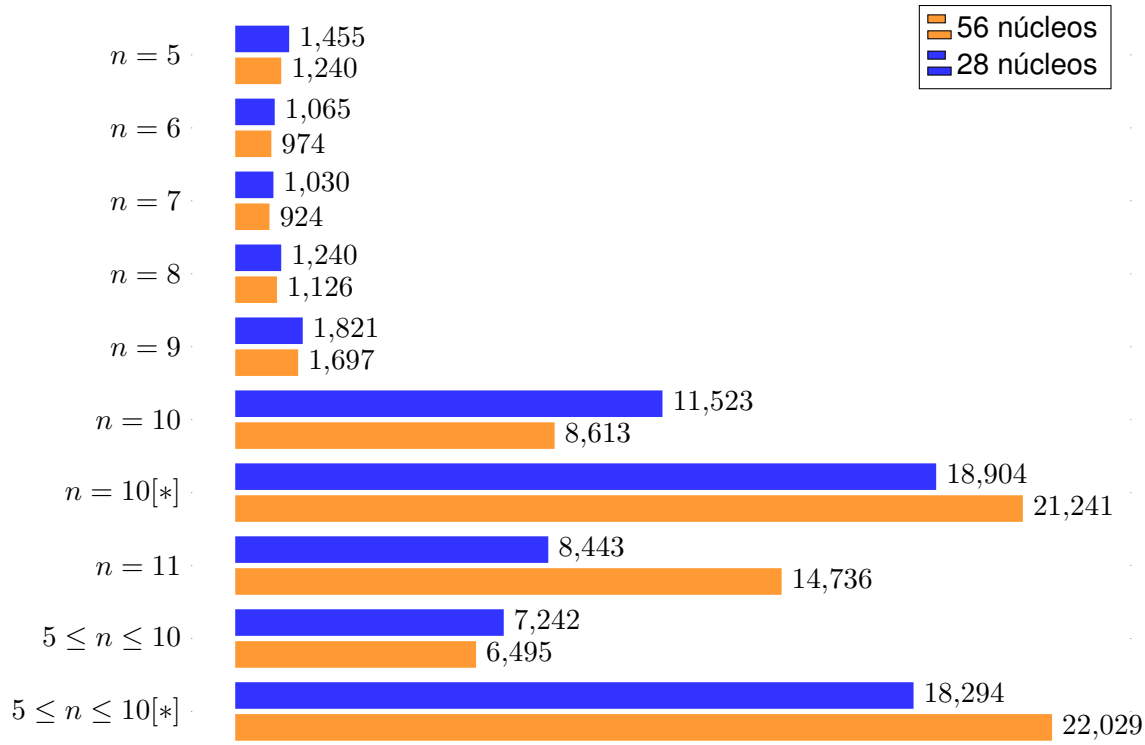


Figura 19 – *Speedup* calculados para grafos de ordem 5 a 11

Outra forma de medir desempenho de aplicação paralela é obtendo medições de eficiência do paralelismo, obtida pela razão entre *speedup* e número de processadores utilizados. Com os dados *speedup* apresentados na Tabela 7, calculamos a Eficiência

para os resultados utilizando 28 e 56 núcleos de processamento.

O Gráfico 20 representa os resultados dos cálculos de Eficiência composto no eixo vertical pela ordem n dos grafos empregados nos testes e no eixo horizontal pela porcentagem da eficiência calculada.

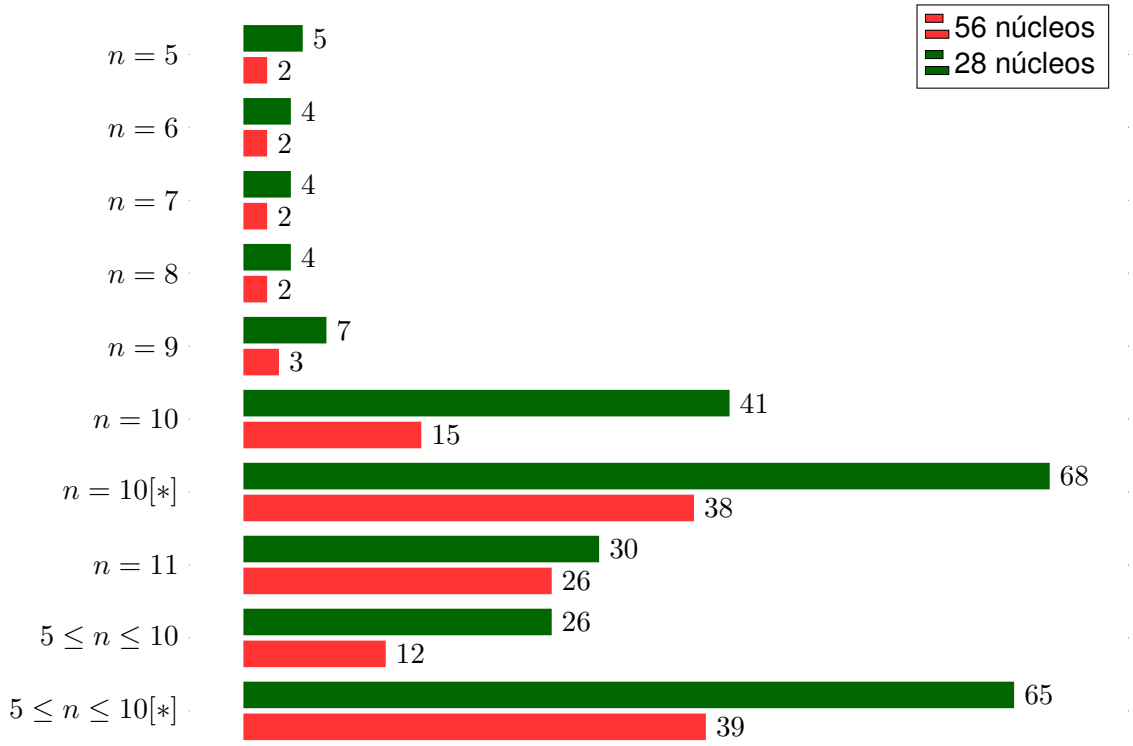


Figura 20 – Eficiência(%) calculada para grafos de ordem 5 a 11

6.3- Análise dos resultados

O código desenvolvido em Python possui ligeiros melhores tempos de execução para testes utilizando grafos de ordem n entre 5 e 7. Mas para testes com $n > 7$, consequentemente com o aumento da base de dados a ser analisada, o código em Java apresenta cada vez melhores resultados ao passo que o valor da ordem n é incrementado aos testes. Como verificado nos resultados apresentados na Tabela 6, para todos grafos de ordem $n = 10$, o tempo de execução em Java é 5,5x mais rápido. O trabalho de Ghebleh et al. [2019] já mencionava a superioridade do Java em cálculos matemáticos em

termos de rapidez perante o Python. O fato de o Spark ser desenvolvido em Scala, que é uma linguagem baseada em Java e utiliza Java Virtual Machine (JVM) em sua execução, também é um fator que contribui para o desempenho superior.

Como o *workflow* desenvolvido em Java obteve com folga o melhor desempenho no resultado do comparativo entre os dois códigos elaborados nesta pesquisa, procedemos para realização nova bateria de testes de execuções utilizando 1, 28 e 56 núcleos de processamento para registro dos tempos médios e desvios-padrão e, posteriormente, cálculos de *speedup* e Eficiência de modo a verificarmos a melhor configuração do ambiente Spark na execução do *workflow*. Analisando a Tabela 7, nota-se que para grafos com ordens entre 5 e 9 os testes utilizando 28 núcleos de processamento obtiveram médias de tempos de execução ligeiramente melhores. No entanto, testes contendo grafos com ordem $n = 10$ e $n = 11$ que tiveram execuções pausadas entre o fim de uma e o início da outra para resfriamento dos processadores e que utilizaram capacidade máxima do servidor Antares (56 núcleos de processamento) obtiveram tempos de execuções melhores.

Analisando o ganho de desempenho com o paralelismos das tarefas no Spark através do *speedup*, verificando o Gráfico 19 podemos observar que para grafos de ordens n entre 5 e 9 o ganho de desempenho foi pouco. Já para os testes com $n = 10$, $n = 11$ e $5 \leq n \leq 10$ o ganho com o paralelismo foi muito produtivo, chegando a execuções 22x mais rápidas para o caso de testes com todos os grafos de ordens $5 \leq n \leq 10$ com o intervalo de resfriamento.

Em termos de eficiência, todos os valores pertinentes aos testes com 28 núcleos se mostraram melhores, como pode ser verificado no Gráfico 20, para todos os testes com grafos de ordem entre 5 e 10. A eficiência calculada para grafos com ordem n entre 5 e 9 demonstra que a execução pode ser realizada com poucos recursos alocados. Para grafos com ordem $n = 10$, a execução com 28 núcleos obteve diferença de *speedup* pequena perante o calculado para 56 núcleos e eficiência acima dos 60% nos testes de modo que podemos concluir que a configuração aplicada foi ideal para o tamanho da base de dados utilizada. Para teste com grafos com $n = 11$, onde a base de dados é 92x maior, configuração com 28 núcleos foi verificada a mais eficiente. Mas como um dos objetivos do portal RioGraphX é a rapidez na obtenção dos resultados, os testes indicaram que a configuração utilizando 56 núcleos é a ideal pois obteve melhor tempo médio de execução e com *speedup* quase 2x mais rápido de diferença.

Essa diferença de quantidade de recursos *versus* eficiência se dá pelo fato que conforme mais recursos (núcleos de processamento) para execução de tarefas paralelas são adicionados ao ambiente, mais o *Driver Program*, responsável pela submissão de tarefas e coleta das informações processadas, tem o trabalho de organizar os resultados recebidos e com isso demandando mais tempo para execução.

7- Conclusão

7.1- Análise retrospectiva

Esta pesquisa apresentou o Portal científico RioGraphX cujo objetivo é fornecer à comunidade científica uma ferramenta computacional de auxílio a pesquisas sobre TEG de modo a propor ou refutar uma conjectura a partir da propriedade de um grafo que o usuário deseja verificar.

A principal contribuição deste trabalho é a construção de uma ferramenta computacional capaz de reforçar conjecturas existentes ou sugerir novas conjecturas a partir de uma análise exaustiva em todos os grafos de uma dada ordem. Neste sentido, o portal RioGraphX é capaz de processar grafos até ordem $n = 11$ com tempo computacional aceitável, possibilidade que não conseguimos verificar em ferramentas similares (Capítulo 2). Além disso, o RioGraphX está disponível *on-line* e sem nenhuma necessidade de instalação local e de programação, o que o torna de grande auxílio para o desenvolvimento das pesquisas em TEG. Outra vantagem da ferramenta em termos de estrutura é sua portabilidade em virtude de estar configurada em um ambiente Docker, caso haja necessidade de replicação ou transferência para outro servidor. Toda a estrutura pode ser configurada facilmente em outro ambiente sem prejuízo das versões das bibliotecas utilizadas e independente da configuração do novo ambiente, seja este utilizando sistema operacional Windows ou Linux.

O desenvolvimento do portal científico resultou em uma interface WEB no *front-end* utilizando um servidor Tomcat e um banco de dados PostgreSQL como apoio. Um *workflow* científico no *back-end* foi construído utilizando a ferramenta Apache Spark com sete etapas bem definidas, de modo que o máximo de tarefas possíveis pudessem ser executadas em um ambiente de processamento paralelo e/ou distribuído.

Para este fim, após diversas pesquisas em ferramentas para construção de *workflows* científicos, foi verificado que o ambiente Apache Spark possui melhores desempenhos e rotinas de controle de falhas, o que assegura a consistência nos resultados.

O Spark possui componentes importantes de apoio como Spark SQL, para traba-

lhos com dados estruturados, e bibliotecas para manipulação e processamento paralelo de grafos como o GraphX e GraphFrames. Após análise das bibliotecas GraphX e GraphFrames, as mesmas foram descartadas para uso no *workflow* pois foram desenvolvidas para manipular grafos gigantes e executar algoritmos como do caminho mínimo, *Prim*, *Dijkstra* e *PageRank*. Além disso, foi verificado que a estrutura de dados *Dataframes* do Spark SQL atende ao objetivo da pesquisa de maneira satisfatória.

Dois códigos-fontes de construção do *workflow* para uso no Spark foram desenvolvidos para execução, um na linguagem Java e outro em Python. Para verificação dos objetivos do Portal RioGraphX, que é consistência dos resultados e agilidade no processamento das informações, foram realizados testes de validação e de desempenho.

O teste de validação, mencionado na Seção 4.5, consistiu na verificação de trabalhos com conjecturas recentes em TEG e sua reprodução no Portal RioGraphX. Este teste foi realizado em ambos os códigos desenvolvidos e todos os cálculos de invariantes realizados foram conferidos com cálculos similares realizados no Portal CoCalc e validados com sucesso.

Os testes de desempenho realizados no Capítulo 6 consistiram na execução de baterias de 10 execuções consecutivas para cada ordem n de grafos dos dois códigos desenvolvidos para o *workflow* e aferição das médias de tempos e desvios-padrão obtidos.

Na análise dos resultados dos testes, o código em Java obteve médias de tempo de execução até 5x mais rápido, para grafos de ordem n entre 5 e 10, em relação ao código em Python, conforme detalhado na tabela 6. Para uma análise mais profunda do código em Java, foram incluídos nos testes de execução todos os grafos de ordem $n = 11$ e duas métricas de desempenho foram verificadas: *speedup* e eficiência. Os Gráficos 19 e 20 apresentam os valores calculados para essas duas métricas mencionadas.

A avaliação dos resultados dos testes de desempenho nos levam a concluir que a quantidade de recursos de processamento a serem alocados devem levar em consideração o tamanho da base de dados, visto que o *Driver Program*, programa responsável gestão das tarefas distribuídas entre as partições do Spark, demanda tempo para organização quando muitos recursos são alocados. Os tempos de execução do portal científico RioGraphX se mostraram satisfatórios, dada a grande carga de informações de grafos processadas e a consistência dos resultados obtidos.

O projeto RioGraphX teve seu início com o discente Carlos Magno Oliveira de Abreu com códigos existentes, disponíveis em `https://github.com/icemagno/spark`,

implementados no ambiente Spark e de grande ajuda no desenvolvimento deste trabalho.

Alguns trabalhos, como o de Ghebleh et al. [2019], citam a incerteza na precisão dos cálculos aritméticos com números reais utilizando bibliotecas do Java e informam que bibliotecas como o Sympy e Numpy do Python possuem melhor acurácia. No *workflow* desenvolvido em Python foram utilizadas tanto Sympy como Numpy e no *workflow* desenvolvido em Java foi utilizada a biblioteca Parsii. Nos testes mencionados na Seção 6.1, foram feitos comparativos dos resultados obtidos e, em todas as análises, os resultados dos cálculos foram idênticos.

O trabalho desenvolvido nesta pesquisa produziu o Artigo *"A Science Gateway to Support Research in Spectral Graph Theory"* [Oliveira et al., 2019] apresentado no 34º Simpósio Brasileiro de Banco de Dados e apresentou o Portal à comunidade científica. O acesso à ferramenta está disponível através do endereço `https://www.riographx.ga` para que pesquisadores de TEG, em todo o mundo, possam obter auxílio em suas pesquisas e proporem novas conjecturas ou explorarem com mais profundidade as existentes.

7.2- Trabalhos futuros

Seguem algumas ideias interessantes para continuidade e aperfeiçoamento deste trabalho:

- Verificação e descoberta de invariantes importantes sendo estudados pela comunidade científica e incorporá-los ao Portal RioGraphX de modo a mantê-lo sempre atualizado;
- A necessidade de uma arquitetura com mais recursos é um passo importante para realização de execuções de grafos com ordem $n > 11$;
- Um modelo de cálculo de alocação dinâmica de recursos do Spark de acordo com a base de dados relacionada também é necessário para aumentar os níveis de eficiência.
- O desenvolvimento de uma versão mobile do Portal seria importante de modo a fornecer mais uma alternativa de acesso ao usuário.

Referências Bibliográficas

Alemi, M., Haghighi, H., and Shahrivari, S. (2017). Ccfinder: using spark to find clustering coefficient in big graphs. The Journal of Supercomputing, 73(11):4683–4710.

Amos, D. and Davila, R. (2017). Grinpy documentation.

Aouchiche, M. and Hansen, P. (2013). Two laplacians for the distance matrix of a graph. Linear algebra and its applications, 439(1):21–33.

Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., et al. (2015). Spark sql: Relational data processing in spark. In Proceedings of the 2015 ACM SIGMOD international conference on management of data, pages 1383–1394. ACM.

Balaban, A. T., Motoc, I., Bonchev, D., and Mekenyan, O. (1983). Topological indices for structure-activity correlations. In Steric effects in drug design, pages 21–55. Springer.

Berlitz, I. H. (2007). Gerador gráfico de relatórios utilizando a classe fpdf. Novo Hamburgo.

Brankov, V., Cvetković, D., Simić, S., and Stevanović, D. (2006). Simultaneous editing and multilabelling of graphs in system newgraph. Publikacije Elektrotehničkog fakulteta. Serija Matematika, pages 112–121.

Brinkmann, G., Coolsaet, K., Goedgebeur, J., and Mélot, H. (2013). House of graphs: a database of interesting graphs. Discrete Applied Mathematics, 161(1-2):311–314.

Bruno, L. (2009). Itext pdf.[online], 2009.

Caporossi, G. (2017). Variable neighborhood search for extremal vertices: The autographix-iii system. Computers & Operations Research, 78:431–438.

Caporossi, G. and Hansen, P. (2000). Variable neighborhood search for extremal graphs: 1 the autographix system. Discrete Mathematics, 212(1-2):29–44.

Chen, X., Zoun, R., Schallehn, E., Mantha, S., Rapuru, K., and Saake, G. (2018). Exploring spark-sql-based entity resolution using the persistence capability. In International Conference: Beyond Databases, Architectures and Structures, pages 3–17. Springer.

Chung, F. R. and Graham, F. C. (1997). Spectral graph theory. Number 92. American Mathematical Soc.

Curry, R., Kiddle, C., and Simmonds, R. (2009). Social networking and scientific gateways. In Proceedings of the 5th Grid Computing Environments Workshop, pages 1–10.

Cvetković, D. (1980). A project for using computers in further development of graph theory. In Theory and Application of graphs (Proc. 4th Internat. Conf. on Theory and Appl. of Graphs, Kalamazoo 1980) edited by Chartrand, G., Alvai, Y., Goldsmith, DL, Lesniak-Foster, L., Lick, DR, pages 285–296.

Cvetković, D. and Simić, S. (2011). Graph spectra in computer science. Linear Algebra and its Applications, 434(6):1545–1562.

Cvetković, D. M., Doob, M., and Sachs, H. (1980). Spectra of graphs: theory and application, volume 87. Academic Pr.

Dave, A., Jindal, A., Li, L. E., Xin, R., Gonzalez, J., and Zaharia, M. (2016). Graphframes: an integrated api for mixing graph and relational queries. In Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, page 2. ACM.

de Oliveira Ribeiro, D. P. and Jones, Á. A. (2020). Estudo em teoria dos grafos e desenvolvimento de ferramentas computacionais para o software newgraph. Simpósio de Pesquisa, Inovação e Tecnologia do Campus Juiz de Fora do IF Sudeste MG, 3.

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. Communications of the ACM, 51(1):107–113.

Diedrich, C. (2015). O que é docker?

Dittrich, J. and Quiané-Ruiz, J.-A. (2012). Efficient big data processing in hadoop mapreduce. Proceedings of the VLDB Endowment, 5(12):2014–2015.

Ellson, J., Gansner, E., Koutsofios, L., North, S. C., and Woodhull, G. (2001). Graphviz—open source graph drawing tools. In International Symposium on Graph Drawing, pages 483–484. Springer.

Estrada, E. (2017). The abc matrix. Journal of Mathematical Chemistry, 55(4):1021–1033.

Estrada, E., Torres, L., Rodriguez, L., and Gutman, I. (1998). An atom-bond connectivity index: modelling the enthalpy of formation of alkanes.

Estrada, E. and Vargas-Estrada, E. (2012). Distance-sum heterogeneity in graphs and complex networks. Applied Mathematics and Computation, 218(21):10393–10405.

Fajtlowicz, S. (1987). On conjectures of graffiti-ii. Congr. Numer, 60:187–197.

Fajtlowicz, S. (1998). Written on the wall. manuscript), Web address: http://math. uh. edu/ siemion.

Ferreira, J. A., Porto, F., Coutinho, R., and Ogasawara, E. (2018). Rumo à otimização de operadores sobre udf no spark. In Anais do XII Brazilian e-Science Workshop. SBC.

Fiedler, M. (1973). Algebraic connectivity of graphs. Czechoslovak mathematical journal, 23(2):298–305.

Ghebleh, M., Kanso, A., and Stevanović, D. (2019). Graph6java: A researcher–friendly java framework for testing conjectures in chemical graph theory. In MATCH Communications in Mathematical and in Computer Chemistry, pages 737–770. MATCH.

Graovac, A. and Ghorbani, M. (2010). A new version of atom-bond connectivity index. Acta Chim. Slov, 57(3):609.

Gregor, D. and Lumsdaine, A. (2005). The parallel bgl: A generic library for distributed graph computations. Parallel Object-Oriented Scientific Computing (POOSC), 2:1–18.

Grijó, R., de Lima, L., Oliveira, C., Porto, G., and Trevisan, V. (2019). Nordhaus–gaddum type inequalities for the two largest laplacian eigenvalues. Discrete Applied Mathematics, 267:176–183.

Gustafson, J. L. (1988). Reevaluating amdahl's law. Communications of the ACM, 31(5):532–533.

Gutman, I. and Zhou, B. (2006). Laplacian energy of a graph. Linear Algebra and its applications, 414(1):29–37.

Hagberg, A., Swart, P., and S Chult, D. (2008). Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

Hall, K. M. (1970). An r-dimensional quadratic placement algorithm. Management science, 17(3):219–229.

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., Shenker, S., and Stoica, I. (2011). Mesos: A platform for fine-grained resource sharing in the data center. In NSDI, volume 11, pages 22–22.

Hogben, L. (2005). Spectral graph theory and the inverse eigenvalue problem of a graph. Electronic Journal of Linear Algebra, 14(1):3.

Hoschek, W. et al. (2004). Colt project.

Huai, Y., Chauhan, A., Gates, A., Hagleitner, G., Hanson, E. N., O'Malley, O., Pandey, J., Yuan, Y., Lee, R., and Zhang, X. (2014). Major technical advancements in apache hive. In Proceedings of the 2014 ACM SIGMOD international conference on Management of data, pages 1235–1246.

Huang, X. and Lin, H. (2019). Signless laplacian eigenvalue problems of nordhaus–gaddum type. Linear Algebra and its Applications, 581:336–353.

Hückel, E. (1931). Quantentheoretische beiträge zum benzolproblem. Zeitschrift für Physik A Hadrons and Nuclei, 70(3):204–286.

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. Computing in science & engineering, 9(3):90–95.

Islam, M. T., Karunasekera, S., and Buyya, R. (2017). Dspark: Deadline-based resource allocation for big data applications in apache spark. In 2017 IEEE 13th International Conference on e-Science (e-Science), pages 89–98. IEEE.

Karau, H., Konwinski, A., Wendell, P., and Zaharia, M. (2015). Learning spark: lightning-fast big data analysis. "O'Reilly Media, Inc.".

Klavžar, S. and Gutman, I. (1996). A comparison of the schultz molecular topological index with the wiener index. Journal of chemical information and computer sciences, 36(5):1001–1003.

Kölling, M., Quig, B., Patterson, A., and Rosenberg, J. (2003). The bluej system and its pedagogy. Computer Science Education, 13(4):249–268.

Levene, R. H., Oblak, P., and Šmigoc, H. (2019). A nordhaus–gaddum conjecture for the minimum number of distinct eigenvalues of a graph. Linear Algebra and its Applications, 564:236–263.

MATLAB (2010). version 7.10.0 (R2010a). The MathWorks Inc., Natick, Massachusetts.

Mavridis, I. and Karatza, H. (2017). Performance evaluation of cloud-based log file analysis with apache hadoop and apache spark. Journal of Systems and Software, 125:133–151.

McKay, B. D. and Piperno, A. (2014). Practical graph isomorphism, {II}. Journal of Symbolic Computation, 60(0):94 – 112.

Mihalić, Z., Veljan, D., Amić, D., Nikolić, S., Plavšić, D., and Trinajstić, N. (1992). The distance matrix in chemistry. Journal of Mathematical Chemistry, 11(1):223–258.

Mladenović, N. and Hansen, P. (1997). Variable neighborhood search. Computers & operations research, 24(11):1097–1100.

Mohar, B., Alavi, Y., Chartrand, G., and Oellermann, O. (1991). The laplacian spectrum of graphs. Graph theory, combinatorics, and applications, 2(871-898):12.

Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., et al. (2018). Ray: A distributed framework for emerging {AI} applications. In 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18), pages 561–577.

Oliveira, C. S., De Lima, L. S., de Abreu, N. M. M., and Kirkland, S. (2010). Bounds on the q-spread of a graph. Linear algebra and its applications, 432(9):2342–2351.

Oliveira, D., Abreu, C. M., Ogasawara, E., Bezerra, E., and de Lima, L. (2019). A science gateway to support research in spectral graph theory. In Anais do XXXIV Simpósio Brasileiro de Banco de Dados, pages 217–222, Porto Alegre, RS, Brasil. SBC.

Pacheco, D., de Lima, L., and Oliveira, C. S. (2020). On the graovac-ghorbani index for bicyclic graphs with no pendant vertices. arXiv preprint arXiv:2005.02141.

Randić, M. (1975). Characterization of molecular branching journal of the american chemical society 97 no 23 6609-6615. Crossref Export Citation.

SageMath, I. (2019). Cocalc collaborative computation online.

Shi, J., Qiu, Y., Minhas, U. F., Jiao, L., Wang, C., Reinwald, B., and Özcan, F. (2015). Clash of the titans: Mapreduce vs. spark for large scale data analytics. Proceedings of the VLDB Endowment, 8(13):2110–2121.

Škrekovski, R., Dimitrov, D., Zhong, J., Wu, H., and Gao, W. (2019). Remarks on multiplicative atom-bond connectivity index. IEEE Access, 7:76806–76811.

Stein, W. (2007). Sage mathematics software. http://www.sagemath.org/.

Stein, W. et al. (2008). Sage: Open source mathematical software. 7 December 2009.

Vasilyev, A. and Stevanović, D. (2014). Mathchem: a python package for calculating topological indices. MATCH Commun. Math. Comput. Chem, 71:657–680.

Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al. (2013). Apache hadoop yarn: Yet another resource negotiator. In Proceedings of the 4th annual Symposium on Cloud Computing, page 5. ACM.

Walt, S. v. d., Colbert, S. C., and Varoquaux, G. (2011). The numpy array: a structure for efficient numerical computation. Computing in Science & Engineering, 13(2):22–30.

Wilf, H. S. (1967). The eigenvalues of a graph and its chromatic number. J. London Math. Soc, 42(1967):330.

Xin, R. S., Gonzalez, J. E., Franklin, M. J., and Stoica, I. (2013). Graphx: A resilient distributed graph system on spark. In First International Workshop on Graph Data Management Experiences and Systems, page 2. ACM.

Átila Arueira Jones and Ribeiro, D. (2020). Graph filter.

A- Exemplo de relatório disponibilizado pelo Portal RioGraphX

Report #id: 196

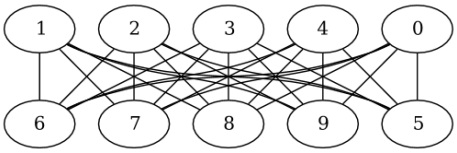
Objective function: (\mu_2 + \overline{\mu_2}) - 2*n - 2
Optimization: Minimum
Only generate triangle free? No
Only generate connected graphs? Yes
Only generate bipartite graphs? No

Report generated in Sat Oct 24 23:04:15 UTC 2020

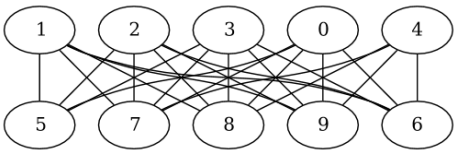
1.

Order: #10

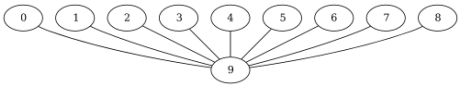
Index	Value
G6 code of the graph	I?B~vrw}?
Order	10
Size	25
Objective function	-12.0



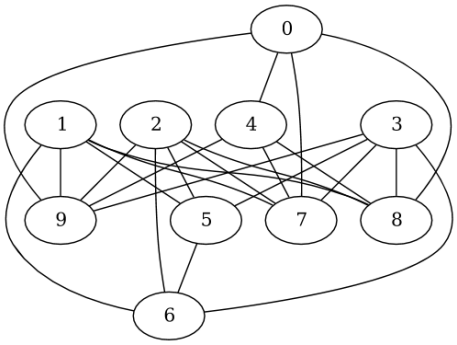
Index	Value
G6 code of the graph	I?Bvvrw}?
Order	10
Size	24
Objective function	-12.0



Index	Value
G6 code of the graph	I??????~w
Order	10
Size	9
Objective function	-12.0



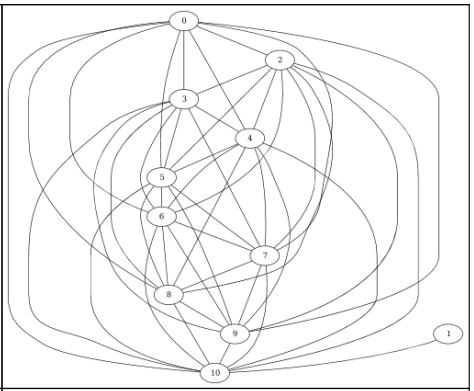
Index	Value
G6 code of the graph	I?`rnrw}?
Order	10
Size	23
Objective function	-12.0



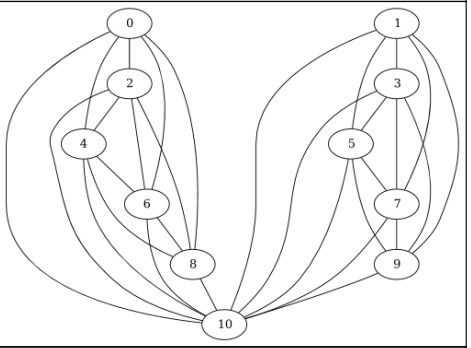
1.

Order: #11

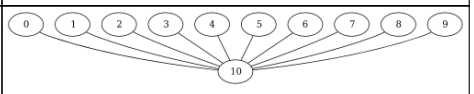
Index	Value
G6 code of the graph	JTm||}~n~~_
Order	11
Size	46
Objective function	-13.0



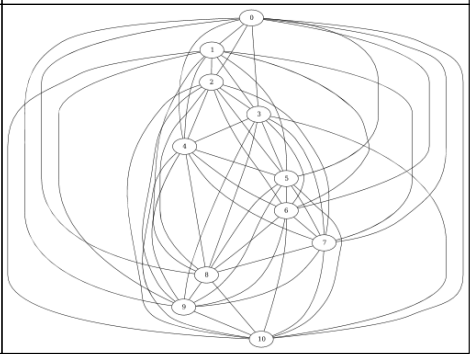
Index	Value
G6 code of the graph	JQhTQiiTV~_
Order	11
Size	30
Objective function	-13.0



Index	Value
G6 code of the graph	J???????F~_
Order	11
Size	10
Objective function	-13.0



Index	Value
G6 code of the graph	J~~~~~~~~~_
Order	11
Size	55
Objective function	-13.0



B- Invariantes de grafos disponíveis no RioGraphX

Definition and Notation for all graph invariants implemented in RioGraphX and its syntax

B.1- Basics

We assume that all graphs $G = (V, E)$ are simple and undirected of order $|V| = n$ and size $|E| = m$.

All eigenvalues are arranged in non-increasing order such that the first index points out to the largest eigenvalue. For instance, for the adjacency matrix its eigenvalues are as follows: $\lambda_1 \geq \cdots \geq \lambda_n$.

B.2- Definitions

- Energy of a graph:

$$E(G) = \sum_{i=1}^{n} |\lambda_i|$$

- Laplacian Energy of a graph:

$$LE(G) = \sum_{i=1}^{n} |\mu_i - 2m/n|$$

- Signless Laplacian Energy of a graph:

$$SLE(G) = \sum_{i=1}^{n} |q_i - 2m/n|$$

- Distance Energy of a graph:

$$E_D(G) = \sum_{i=1}^{n} |\rho_i|$$

- Distance Laplacian Energy of a graph:

$$LE_D(G) = \sum_{i=1}^{n} \left| \gamma_i - \frac{1}{n} \sum_{i=1}^{n} D_i \right|,$$

  where D_i is the sum of distances between vertex v_i and the other vertices of G, that is, the i-th entry of the transmission matrix.

- Distance Signless Laplacian Energy of a graph:

$$SLE_D(G) = \sum_{i=1}^{n} \left| \alpha_i - \frac{1}{n} \sum_{i=1}^{n} D_i \right|,$$

  where D_i is the sum of distances between vertex v_i and the other vertices of G, that is, the i-th entry of the transmission matrix.

B.3- Syntax of each invariant

- Largest i-th adjacency eigenvalue: `\lambda_i`
- Largest i-th Laplacian eigenvalue: `\mu_i`
- Largest i-th signless Laplacian eigenvalue: `q_i`
- Largest i-th Distance eigenvalue: `\rho_i`
- Largest i-th Laplacian Distance eigenvalue: `\gamma_i`
- Largest i-th Distance signless Laplacian eigenvalue: `\alpha_i`
- Largest i-th adjacency eigenvalue of the graph complement: `\overline{\lambda}_i`
- Largest i-th Laplacian eigenvalue of the graph complement: `\overline{\mu}_i`
- Largest i-th signless Laplacian eigenvalue of the graph complement: `\overline{q}_i`

- Largest i-th Distance eigenvalue of the graph complement: `\overline{\rho}_i`
- Largest i-th Laplacian Distance eigenvalue of the graph complement: `\overline{\gamma}_i`
- Largest i-th Distance signless Laplacian eigenvalue of the graph complement: `\overline{\alpha}_i`
- Chromatic number: `\chi`
- Chromatic number of the graph complement: `\overline{\chi}`
- Clique number: `\omega`
- Clique number of the graph complement: `\overline{\omega}`
- Largest i-th degree of the graph: `d_i`
- Energy of a graph: E
- Laplacian Energy of a graph: LE
- Signless Laplacian Energy of a graph: SLE
- Distance Energy of a graph: E_D (displays as E_D)
- Distance Laplacian Energy of a graph: LE_D (displays as LE_D)
- Distance Signless Laplacian Energy of a graph: SLE_D (displays as SLE_D)
- Atom-Bond connectivity index of a graph: ABC (displays as ABC)
- Atom-Bond connectivity index of a graph complement: `\overline{ABC}`
- Largest i-th eigenvalue of the ABC matrix of a graph: `\beta_i`
- Largest i-th eigenvalue of the ABC matrix of a graph complement: `\overline{\beta}_i`
- Graovac-Ghorbani index of a graph: ABC_{GG} (displays as ABC_{GG})
- Graovac-Ghorbani index of a graph complement: `\overline{ABC}_{GG}` (displays as $\overline{ABC}_{GG}$)

C- Tabelas contendo todos os tempos de execução dos testes

Rodada/Ordem	$n = 5$	$n = 6$	$n = 7$	$n = 8$	$n = 9$	$n = 10$	$n = 11$	$5 <= n <= 10$	$n = 10$(*)	$5 <= n <= 10$(*)
t_1	14,350	10,789	10,987	16,042	54,022	115,634	8.588,268	206,501	112,999	117,444
t_2	11,122	11,056	10,716	14,507	57,430	106,714	8.624,899	196,716	114,513	108,676
t_3	11,124	10,593	10,848	14,146	57,151	115,753	8.616,708	180,534	114,298	117,235
t_4	10,633	10,882	10,855	15,825	61,655	130,373	8.493,361	339,320	110,525	107,755
t_5	11,129	10,550	11,318	14,565	54,533	264,635	8.504,462	530,102	113,285	108,577
t_6	10,689	10,720	10,691	13,963	55,932	394,128	8.359,763	522,807	117,456	120,752
t_7	10,897	10,830	10,967	15,591	56,981	434,728	8.450,543	487,712	122,314	110,6
t_8	11,364	10,767	11,304	13,691	56,558	450,710	8.419,705	429,597	120,63	111,89
t_9	10,914	10,729	10,639	13,973	56,630	394,070	8.379,030	457,167	118,483	113,974
t_10	10,977	11,031	10,620	14,463	56,946	476,495	8.435,686	462,969	124,609	107,506
Média	11,320	10,795	10,895	14,677	56,784	288,324	8.487,243	381,343	116,911	112,441
Desvio-padrão	$\pm 1,087$	$\pm 0,165$	$\pm 0,254$	$\pm 0,841$	$\pm 2,050$	$\pm 157,650$	$\pm 95,904$	$\pm 139,421$	$\pm 4,557$	$\pm 4,697$

Tabela 8 – Tempos de execução do *workflow* em código Java utilizando 56 núcleos de processamento

Rodada/Ordem	$n = 5$	$n = 6$	$n = 7$	$n = 8$	$n = 9$	$n = 10$	$n = 11$	$5 <= n <= 10$	$n = 10$(*)	$5 <= n <= 10$(*)
t_1	10,202	9,702	9,981	13,359	53,308	131,627	15.357,231	132,783	136,871	132,726
t_2	9,951	9,908	9,764	14,759	52,511	131,475	16.172,627	218,670	133,219	138,187
t_3	9,521	10,278	10,192	13,380	50,332	152,582	14.616,288	324,386	135,768	138,090
t_4	9,318	9,916	10,094	12,380	51,185	188,039	14.061,681	410,812	128,871	130,659
t_5	10,088	9,971	9,577	14,074	52,646	235,680	14.279,555	393,300	128,245	133,600
t_6	9,988	9,930	9,524	13,418	53,412	241,792	14.394,987	370,234	132,989	136,790
t_7	9,397	9,720	10,101	13,116	54,431	256,560	13.295,288	404,403	127,798	136,348
t_8	9,638	9,642	9,540	12,729	56,823	247,441	14.032,079	387,008	128,911	135,117
t_9	9,286	10,037	9,541	13,460	56,658	266,371	12.658,314	382,486	128,523	142,482
t_10	9,110	9,648	9,407	12,537	47,705	303,560	14.425,895	396,185	132,430	129,979
Média	9,650	9,875	9,772	13,321	52,901	215,513	14.329,395	342,027	131,363	135,398
Desvio-padrão	$\pm 0,382$	$\pm 0,201$	$\pm 0,293$	$\pm 0,712$	$\pm 2,769$	$\pm 60,511$	$\pm 976,064$	$\pm 93,066$	$\pm 3,328$	$\pm 3,805$

Tabela 9 – Tempos de execução do *workflow* em código Java utilizando 28 núcleos de processamento

Rodada/Ordem	$n = 5$	$n = 6$	$n = 7$	$n = 8$	$n = 9$	$n = 10$	$n = 11$	$5 <= n <= 10$
t_1	26,729	11,678	10,853	17,737	87,772	2.504,002	130.016,079	2.490,910
t_2	13,160	10,260	10,284	15,068	82,072	2.492,176	128.397,894	2.379,591
t_3	13,779	10,197	10,460	15,783	93,592	2.493,310	122.847,634	2.458,931
t_4	12,064	10,056	10,111	16,801	118,074	2.459,165	127.953,895	2.456,305
t_5	11,580	9,610	9,207	17,302	124,103	2.455,967	124.127,693	2.420,212
t_6	12,247	10,720	10,369	15,246	106,455	2.490,505	126.710,380	2.443,431
t_7	13,899	10,564	9,772	18,664	103,908	2.463,583	122.015,169	2.653,431
t_8	11,427	9,502	10,369	17,517	87,882	2.485,524	126.604,751	2.488,068
t_9	11,218	10,892	9,096	16,408	83,343	2.501,952	118.885,269	2.487,993
t_10	14,286	11,648	10,102	14,712	76,352	2487,06	123.149,298	2.491,112
Média	14,039	10,513	10,062	16,524	96,355	2.483,324	125.070,806	2.476,998
Desvio-padrão	$\pm 4,593$	$\pm 0,749$	$\pm 0,555$	$\pm 1,305$	$\pm 16,081$	$\pm 17,464$	$\pm 3.431,886$	$\pm 71,770$

Tabela 10 – Tempos de execução do *workflow* em código Java utilizando 1 núcleo de processamento

Rodada/Ordem	$n = 5$	$n = 6$	$n = 7$	$n = 8$	$n = 9$	$n = 10$	$5 <= n <= 10$
t_1	7,372	7,265	8,804	33,236	338,246	1.365,986	1.364,928
t_2	7,065	7,155	8,822	32,289	335,861	1.636,153	1.428,015
t_3	6,959	7,095	8,931	32,329	337,649	1.634,730	1.656,167
t_4	6,815	7,056	8,840	32,278	337,052	1.575,689	1.656,296
t_5	6,938	6,987	8,830	32,401	338,478	1.338,599	1.684,864
t_6	6,918	7,091	8,981	32,162	335,270	1.594,136	1.687,746
t_7	6,824	7,079	8,809	32,312	340,299	1.601,935	1.677,692
t_8	6,853	7,036	8,727	32,428	338,027	1.604,954	1.678,977
t_9	6,678	7,090	8,965	32,521	344,270	1.629,538	1.664,177
t_10	6,802	7,060	8,835	32,172	339,032	1.614,666	1.690,379
Média	6,922	7,091	8,854	32,413	338,418	1.559,639	1.618,924
Desvio-padrão	$\pm 0,190$	$\pm 0,075$	$\pm 0,080$	$\pm 0,309$	$\pm 2,522$	$\pm 111,084$	$\pm 118,810$

Tabela 11 – Tempos de execução do *workflow* em código Python utilizando 56 núcleos de processamento