Universidade Federal de Ouro Preto
Instituto de Ciências Exatas e Aplicadas
Departamento de Computação e Sistemas

Programação Visual para Sistemas de Mineração de padrões em grafos

Ricardo Rodiani da Silva

TRABALHO DE CONCLUSÃO DE CURSO

ORIENTAÇÃO:
Vinícius Vitor dos Santos Dias

Março, 2023
João Monlevade–MG

Ricardo Rodiani da Silva

Programação Visual para Sistemas de Mineração de padrões em grafos

Orientador: Vinícius Vitor dos Santos Dias

Monografia apresentada ao curso de Engenharia de Computação do Instituto de Ciências Exatas e Aplicadas, da Universidade Federal de Ouro Preto, como requisito parcial para aprovação na Disciplina "Trabalho de Conclusão de Curso II".

Universidade Federal de Ouro Preto
João Monlevade
Março de 2023

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E APLICADAS
DEPARTAMENTO DE COMPUTACAO E SISTEMAS

FOLHA DE APROVAÇÃO

Ricardo Rodiani da Silva

Programação visual para sistemas de mineração de padrões em grafos

Monografia apresentada ao Curso de Engenharia da Computação da Universidade Federal
de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Engenharia da Computação

Aprovada em 31 de março de 2023

Membros da banca

Mestre - Vinícius Vitor dos Santos Dias - Orientador (Universidade Federal de Ouro Preto)
Doutor - Carlos Henrique Gomes Ferreira - (Universidade Federal de Ouro Preto)
Doutor - Luiz Carlos Bambirra Torres - (Universidade Federal de Ouro Preto)

Vinícius Vitor dos Santos Dias, orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 31/03/2023

Documento assinado eletronicamente por **Vinícius Vitor dos Santos Dias**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 31/03/2023, às 09:53, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **0502083** e o código CRC **51804CF8**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: (31)3808-0819 - www.ufop.br

Dedico este trabalho a todas as pessoas que, direta ou indiretamente, contribuíram para minha formação acadêmica e pessoal. Aos meus pais, por serem minha base e meu maior incentivo ao longo de toda minha jornada. Aos meus familiares, amigos e professores, por terem me apoiado e me ajudado a alcançar meus objetivos. E a meus irmãos da República DuBodi que sempre foram a minha segunda família nessa jornada da graduação. Que este trabalho possa ser uma pequena contribuição para a evolução do conhecimento em nossa área e que possa inspirar novas ideias e iniciativas.

Agradecimentos

Gostaria de expressar minha gratidão a todas as pessoas e instituições que me ajudaram durante a elaboração do meu TCC.

Em primeiro lugar, gostaria de agradecer ao meu orientador, Vinícius Vitor dos Santos Dias, por sua orientação, conhecimento e paciência durante todo o processo de elaboração do trabalho. Suas sugestões, comentários e feedbacks foram essenciais para o desenvolvimento de um trabalho de qualidade e para minha formação acadêmica como um todo.

Também gostaria de agradecer à República Dubodi, onde morei durante o período de elaboração do TCC. A convivência com meus colegas de república foi fundamental para manter meu equilíbrio emocional e me permitiu focar nos estudos com mais tranquilidade.

Agradeço ainda aos meus familiares, amigos e professores que me apoiaram e me incentivaram ao longo de todo o processo, sempre me encorajando a continuar em busca dos meus objetivos.

Por fim, gostaria de agradecer à Universidade Federal de Ouro Preto (UFOP), pela oportunidade de realizar meus estudos em uma instituição de excelência e renome nacional. A UFOP, através de seus professores, bibliotecas e laboratórios, proporcionou a infraestrutura necessária para o desenvolvimento deste trabalho e contribuiu significativamente para a minha formação acadêmica.

A todos vocês, meu muito obrigado!

“Science is more than a body of knowledge; it is a way of thinking.”

— Carl Sagan (1934 – 1996),
in: The Demon-Haunted World: Science as a Candle in the Dark.

Resumo

Considerando o problema de usabilidade e curva de aprendizado em ferramentas de processamento de grandes quantidades de dados, foi proposto o desenvolvimento de uma interface de programação visual, com o objetivo de tornar o desenvolvimento de aplicações de mineração de padrões em grafos mais intuitivo e simples para usuários não-programadores.

A implementação da interface de programação visual foi realizada sobre uma arquitetura que foi definida, arquitetura essa que permite a criação de programas utilizando blocos gráficos. Foram desenvolvidos diversos blocos específicos para a manipulação de dados e processamento de informações em grandes volumes.

Ao final do desenvolvimento, a ferramenta foi submetida a testes de em diferentes casos de usos e também foi avaliada de forma quantitativa para avaliar o tempo de execução de cada algoritmo. Os resultados indicaram que a interface de programação visual é capaz de reduzir a curva de aprendizado e tornar o desenvolvimento de aplicações de MPG mais acessível para usuários sem formação na área de programação.

Dessa forma, a interface de programação visual apresentou-se como uma solução viável para o problema de usabilidade e curva de aprendizado em ferramentas de processamento de grandes quantidades de dados, ampliando o leque de usuários capazes de utilizar essas ferramentas e promovendo uma democratização do acesso a tecnologias de análise de dados.

Palavras-chaves: usabilidade. curva de aprendizado. interface de programação visual. processamento de informações. usuários não-programadores. democratização do acesso a tecnologias. aplicações de MPG. Mineração de padrões em grafos. Fractal. Blockly.

Abstract

Considering the problem of usability and learning curve in tools for processing large amounts of data, the development of a visual programming interface was proposed, with the aim of making the development of applications more intuitive and simple for non-programming users.

The implementation of the visual programming interface was performed on an architecture that allows the creation of programs using graphical blocks. Several specific blocks were developed for data manipulation and information processing in large volumes.

At the end of the development, the tool was submitted to tests in different use cases and was also evaluated quantitatively to assess the execution time of each algorithm. The results indicated that the visual programming interface is capable of reducing the learning curve and making the development of Graph Pattern Mining applications more accessible to users without programming backgrounds.

Thus, the visual programming interface presented itself as a viable solution to the problem of usability and learning curve in tools for processing large amounts of data, expanding the range of users capable of using these tools and promoting democratization of access to data analysis technologies.

Keywords: usability. learning curve. visual programming interface. information processing. non-programming users. democratization of access to technologies. Graph Pattern Mining. Fractal. Blockly.

Lista de ilustrações

Figura 1 – Grafo Exemplo . . . 13
Figura 2 – Grafo de entrada . . . 15
Figura 3 – Possível nova conexão entre José e Roberta . . . 15
Figura 4 – Conexões já conhecidas . . . 15
Figura 5 – Projeto Milo . . . 20
Figura 6 – IoT . . . 20
Figura 7 – Arquitetura do Projeto . . . 27
Figura 8 – Exemplo de aplicação *Blockly* . . . 29
Figura 9 – Bloco de entrada . . . 30
Figura 10 – Input . . . 32
Figura 11 – Expand . . . 34
Figura 12 – Aggregate . . . 35
Figura 13 – Filter . . . 37
Figura 14 – Tela Inicial . . . 38
Figura 15 – Bloco input . . . 39
Figura 16 – Bloco input + expand . . . 39
Figura 17 – Motifs, k = 3 . . . 40
Figura 18 – Subgrafos resultantes . . . 40
Figura 19 – Geração de código . . . 41
Figura 20 – Solução montada na Interface . . . 49
Figura 21 – Uma amostra dos subgrafos resultantes . . . 50
Figura 22 – Solução montada na Interface . . . 51
Figura 23 – Relação Políticos x Companhias . . . 51
Figura 24 – Solução montada na Interface . . . 53
Figura 25 – Comparação entre 3 pares de áreas distintas . . . 54
Figura 26 – Comparação Motifs . . . 55
Figura 27 – Comparação Cliques . . . 56
Figura 28 – Passo 1 Cliques 3 . . . 61
Figura 29 – Passo 2 Cliques 3 . . . 61
Figura 30 – Passo 3 Cliques 3 . . . 61
Figura 31 – Passo 1 Motifs 3 . . . 62
Figura 32 – Passo 2 Motifs 3 . . . 62
Figura 33 – Passo 3 Motifs 3 . . . 62

Lista de tabelas

Tabela 1 – Grafos utilizados 47
Tabela 2 – Motifs, no terminal 55
Tabela 3 – Motifs, na interface 55
Tabela 4 – Cliques, no terminal 56
Tabela 5 – Cliques, na interface 56

Sumário

1 INTRODUÇÃO . . . **12**
1.1 Conceitos e definições preliminares . . . **12**
1.2 Definição do problema . . . **16**
1.3 Objetivos Gerais . . . **16**
1.4 Objetivos Específicos . . . **17**
1.5 Metodologia . . . **17**
1.6 Organização do trabalho . . . **18**

2 REVISÃO BIBLIOGRÁFICA . . . **19**
2.1 Programação Visual . . . **19**

3 DESENVOLVIMENTO . . . **22**
3.1 *Fractal* . . . **22**
3.2 Arquitetura do Projeto . . . **26**
3.3 *Frontend* . . . **28**
3.3.1 Interação com usuário . . . 30
3.3.2 Geração de código . . . 30
3.3.3 Geração de código para MPG . . . 32
3.3.4 Geração de código em etapas . . . 38
3.3.5 Comunicação com o *Backend* . . . 41
3.4 *Backend* . . . **42**
3.4.1 Servidor de REST API . . . 43
3.4.2 Execução de aplicações de MPG . . . 44

4 RESULTADOS . . . **47**
4.1 Estudo de casos . . . **47**
4.1.1 Citeseer . . . 48
4.1.2 Facebook . . . 50
4.1.3 Mico . . . 52
4.2 Análise do custo adicional . . . **53**
4.2.1 Motifs com k vértices . . . 54
4.2.2 Cliques com k vértices . . . 56

5 CONCLUSÃO . . . **57**

REFERÊNCIAS 58

APÊNDICES 60

1 Introdução

Uma linguagem de programação visual (VPL) é qualquer sistema que permita que usuários usem majoritariamente elementos visuais para criação de novos programas funcionais, minimizando a escrita textual de determinado programa. (JOST et al., 2014) Esses elementos visuais podem ser dos mais variados tipos, porém é comum o uso de caixas, setas, figuras, que podem simbolizar blocos lógicos, ideias, comandos, ações, etc.

A programação visual torna a criação de novos programas mais intuitivo, diminuindo a curva de aprendizado e até mesmo substituindo por completo a codificação manual. Existem inúmeros projetos que visam simplificar a tarefa de programar. Como exemplo, podemos citar o *Scratch*[1], projeto que visa tornar mais acessível o ensino de pensamento lógico, computacional, criativo para um público infantil. O problema nesse caso é como ensinar programação para um público infantil fora dos meios tradicionais que são teóricos e extremamente abstratos para esse público. A solução então é uma plataforma de ensino que simula uma experiência do tipo "Lego" onde cada bloco visual tem uma função diferente que representa um comando específico e esses blocos podem ser combinados como forma de projetar qualquer tipo de solução que pode ser codificada.

Assim como a programação visual pode resolver o problema de ensino de programação para um público infantil, o conceito também pode expandido para outras áreas. Sistemas de programação visual podem agregar como um facilitador no desenvolvimento de novas soluções. Por isso, vamos considerar para essa monografia problemas de mineração de padrôes em grafos, para os quais não existe solução que envolva a programação visual

A mineração de padrões em grafos (MPG) se refere a uma classe de algoritmos que buscam o processamento de subgrafos extraídos de um único grafo de entrada. A partir dessa entrada o objetivo é gerar subgrafos conectados com determinadas características explicitadas pelo usuário.

É possível observar que a MPG não se trata de uma tarefa trivial computacionalmente e sua codificação pode ser complexa, principalmente se considerarmos que os usuários que os usuários que seriam mais beneficiados com a ferramenta têm conhecimento básico sobre programação ou até mesmo nenhum.

1.1 Conceitos e definições preliminares

Para o entendimento completo deste trabalho devemos fazer uma breve explicação de conceitos e definições importantes. A seguir, definimos o formato do grafo de entrada

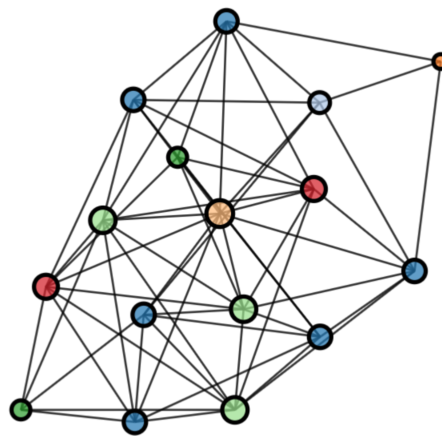
[1] https://scratch.mit.edu/

utilizado como entrada, discutimos a categoria de algoritmos que assumimos neste trabalho (Mineração de Padrões em Grafos, ou MPG) e terminamos com a definição do problema abordado e alvo de nossas contribuições.

Grafo de entrada

Vamos imaginar que exista um grafo G (Figura 1) que represente a amizade ou inimizade entre diferentes tribos (READ, 1954), o mesmo possui vértices e arestas não-direcionadas, podendo conter múltiplos rótulos (cores, numeração, texto, etc.) que podem se repetir.

Figura 1 – Grafo Exemplo



Fonte: (READ, 1954)

Com essa representação que vemos na Figura 1, temos uma estrutura conhecida como **Grafo**, que em nosso exemplo apresentado, representa a amizade ou inimizade entre diferentes tribos de um local. Podemos então definir que um **grafo** é um conjunto de pontos, que são conhecidos como vértices e arestas que representam a conexão entre pares de pontos. Nesse caso, as tribos são os pontos coloridos e uma amizade ou inimizade mútua entre duas tribos é representada por uma linha que conecta esses dois pontos, daqui em diante, chamaremos essa conexão de **aresta** e os pontos de **vértices**.

No contexto de mineração de padrões em grafos utilizando o exemplo de uma representação de amizades e inimizades entre tribos de um local, um **subgrafo** seria uma estrutura menor presente dentro do grafo maior que representa as conexões entre as tribos. Por exemplo, pode haver um **subgrafo** que representa as conexões entre as tribos da mesma região geográfica.

Já um **padrão** em um grafo, neste contexto, pode ser identificado como uma subestrutura que se repete em diferentes partes do grafo. Por exemplo, pode haver um

padrão de conexões em que uma tribo é conectada a duas tribos diferentes, que também estão conectadas entre si. Esse **padrão** pode se repetir em diferentes regiões geográficas dentro do grafo. A identificação desses padrões pode ajudar a entender melhor a estrutura e as relações dentro do grafo.

Temos também o conceito de **grafo não direcionado**, um tipo de grafo em que as arestas não possuem uma direção ou orientação específica. Isso significa que, para cada par de vértices adjacentes, a relação de conectividade é bidirecional, ou seja, uma aresta entre os vértices representa uma conexão em ambos os sentidos. Em nosso exemplo o fato da amizade ou inimizade ser uma reação mútua implica que as arestas sejam não direcionadas, e um grafo que tenha somente arestas desse tipo podem ser classificado com grafo não direcionado.

Outro conceito importe é o de **isomorfismo**, no contexto de tribos, por exemplo, pode haver dois grafos diferentes que representam as conexões entre as tribos, mas que possuem a mesma estrutura de conexões e relações entre os vértices. Nesse caso, esses dois grafos seriam considerados isomorfos. Isso pode ser útil em diversas aplicações, como a identificação de comunidades ou grupos de tribos que possuem as mesmas conexões e relações dentro do grafo.

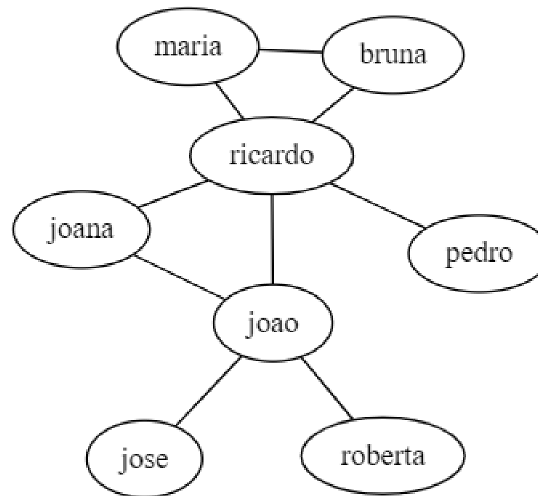
Mineração de padrões em grafos (MPG)

Tendo o conceito de Grafo já definido, outro conceito que será necessário durante o desenvolvimento da monografia é o da mineração de padrões em grafos ou MPG, que se refere ao processo de processar e obter subgrafos resultantes, e a partir desta entrada podemos estabelecer padrões diferentes e relações entre subgrafos possíveis. O processo de MPG pode ser utilizado em diversos contextos como identificação de motifs (padrões que se destacam por sua ocorrência) em redes biológicas de interações entre proteínas (MILO et al., 2002), detecção de fraude em redes financeiras (HOFFMAN; KRASLE, 2015), análise e caracterização de redes sociais (UGANDER; BACKSTROM; KLEINBERG, 2013), entre outras.

Vamos considerar o Grafo representado pela Figura 2 como nossa entrada, nesse caso podemos tratar o Grafo a seguir como uma rede de amizades entre pessoas que se conhecem, uma rede social. Sendo cada aresta desse grafo uma conexão mútua, ou seja, ambas pessoas têm amizade um com o outro.

Num contexto de uma rede social, existe uma grande probabilidade de ocorrer amizades em subgrafos onde exista somente conexões entre 2 arestas (EASLEY; KLEINBERG, 2010), restando uma conexão a ser feita ainda. Isso faz todo sentido, já que as pessoas têm muito mais chance de fazer amizades com amigos de seus amigos do que com completo desconhecidos. Na Figura 3 e Figura 4 podemos observar casos em que ainda é

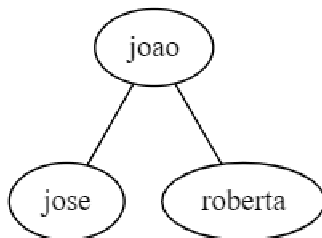
Figura 2 – Grafo de entrada



Fonte: Produzido pelo autor

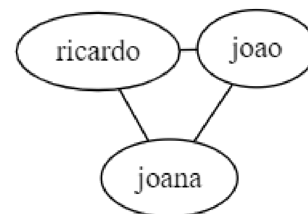
possível existir mais uma conexão entre pessoas, e outra onde todas as conexões possíveis já existem, considerando apenas esses subgrafos.

Figura 3 – Possível nova conexão entre José e Roberta



Fonte: Produzido pelo autor

Figura 4 – Conexões já conhecidas



Fonte: Produzido pelo autor

Agora podemos imaginar que existem n-subgrafos com esses 2 Padrões em um Grafo original, caso nosso objetivo como rede social seja sugerir amizades novas, podemos procurar por subgrafos do padrão que vemos na Figura 3. Em um grafo pode existir uma quantidade exponencialmente grande de padrões diferentes e se isso for utilizado para a recomendação de novas amizades, precisaremos de algoritmos eficientes para extrair esses subgrafos. A MPG surge então com uma classe de algoritmos que podem resolver esse e outros problemas envolvendo Grafos e grandes quantidade de dados disponíveis. Para ter um algoritmo eficiente de sugestão de amizade, a rede social poderia filtrar o subgrafo de conexões de determinado usuário e eliminar subgrafos do padrão a Figura 4, procurando por subgrafos semelhantes a Figura 3 e sugerindo a amizade entre os dois vértices desconhecidos até então, porém com um conhecido em comum.

1.2 Definição do problema

A programação de ferramentas que processam grandes quantidades de dados cada vez mais está se tornando uma tarefa não mais restrita a profissionais com formação da área, mas a usuários finais que apenas desejam processar alguma informação, construir sistemas simples, elaborar algoritmos, automatizar tarefas. Não importando a área de atuação e formação desse usuário é necessário que cada vez mais existem alternativas com uma curva de aprendizado fácil e intuitiva. Uma interface gráfica robusta é capaz de solucionar o problema da usabilidade e curva de aprendizado se ela for bem projetada, podendo também ampliar o leque de usuários capazes de utilizar a ferramenta.

Em problemas de MPG não existem interfaces gráficas com elementos visuais que atinjam o mesmo resultado de ferramentas já consolidadas para esse processamento. O usuário dessas ferramentas deve saber configurar ambientes de programação, assim como dependências de pacotes, instalação de compiladores das linguagens utilizadas assim como saber interpretar a lógica de programação e codificar seu próprio programa. Atualmente esse tipo de conhecimento só é ensinado para cursos específicos de graduação que envolvem computação e programação, tornando assim inviável a utilização de ferramentas poderosas por usuários não-programadores.

No caso de um usuário não-programador atualmente para utilizar uma ferramenta de MPG tipicamente ele irá instalar e configurar o ambiente de programação, variáveis de sistema, etc. Entender a linguagem que a ferramenta foi construída, sua sintaxe, estruturação. Assim como conseguir desenvolver seu próprio código tendo um conhecimento nulo ou básico de lógica de programação. E em muitos casos ainda terá que gerar os executáveis que rodam o programa desenvolvido, conhecimento também desconhecido por usuários não-programadores.

Fica evidente que dessa forma a ferramenta se torna restrita a apenas pessoas ligadas a cursos de computação e sistemas de informação ou entusiastas da área. Surge então a programação visual como uma alternativa para conectar as ferramentas de MPG com usuários não programadores, através de uma boa interface gráfica, intuitiva e usável o leque de usuários aumenta. O objetivo deste trabalho é então, projetar e desenvolver um sistema que torne o desenvolvimento de aplicações MPG mais intuitivo e simples de ser realizado por usuários não-programadores utilizando a programação visual.

1.3 Objetivos Gerais

Para solucionar esse problema devemos então cumprir alguns requisitos ao se projetar uma interface que seja capaz de resolver problemas de MPG.

- Definir qual ferramenta de MPG será utilizada;

- Estudar a ferramenta, estudo de casos e operadores principais;
- Analisar a bibliografia existente no contexto de Programação Visual;
- Projetar uma interface que contenha elementos visuais;
- Implementar e testar com o estudo de casos;

1.4 Objetivos Específicos

Para resolver cada item geral, devemos tomar os seguintes passos:

- Detalhar os requisitos que a ferramenta deve atender, consultar as referências e fazer a escolha com base em critérios previamente definidos.
- Detalhar como a ferramenta se comporta, analisar exemplos, criar um estudo de casos para solucionar problemas previamente propostos, assim como avaliar a sintaxe e construção necessária para a utilização da ferramenta.
- Consultar o referencial existente para entender as soluções que foram propostas tendo como base a programação visual e definir o que podemos usar e como podemos usar, assim como criar novas soluções e adaptar ao nosso contexto.
- Projetar nosso próprio sistema com as práticas que podem ser aproveitadas em nosso contexto, é necessário também desenvolver uma solução nova caso seja necessário levando em conta pontos que foram observados em etapas anteriores.
- Desenvolver a solução levando em conta o estudo de casos.

1.5 Metodologia

A solução então será desenvolvida da seguinte forma:

- Análise bibliográfica sobre Programação Visual: Buscar estudos de ferramentas que tiveram como solução principal a programação visual, e tomar conhecimento do contexto e como foi executado.
- Estudo da ferramenta de MPG: Com uma ferramenta definida, é necessário detalhar a mesma e entender o funcionamento em completo, assim como criar um estudo de casos para os problemas que desejamos resolver (MPG).
- Construir o *Frontend* de nossa aplicação: Aqui devemos focar o desenvolvimento na parte visual do sistema, elementos gráficos e interatividade do usuário com o mesmo.

- Construir o *Backend* de nossa aplicação: Aqui é necessário desenvolver e criar a "ponte" entre a nossa ferramenta escolhida e nossa interface visual.
- Validar o projeto a partir de estudo de casos: Uma vez finalizado o desenvolvimento, os casos usos previamente descritos deverão apresentar resultados equivalentes na interface desenvolvida, devemos analisar também questões de desempenho e usabilidade da solução feita.

1.6 Organização do trabalho

O restante deste trabalho é organizado da seguinte maneira. O Capítulo 2 trata da revisão bibliográfica sobre Programação Visual, MPG, assim como quais ferramentas serão utilizadas para o desenvolvimento deste trabalho. O Capítulo 3 trata como foi feito a escolha de nossa ferramenta MPG, assim como os casos possíveis e como podemos usá-la. Aqui também será descrito como foi o processo de desenvolvimento da interface de programação visual, breve explicação da construção da interface e bibliotecas ou *frameworks* que foram necessários serem utilizados. No Capítulo 4 colocaremos a prova nossos casos de testes que servirão para analisar se a interface cumpriu os objetivos específicos. E ao final no Capítulo 5 se dá a conclusão dos estudos feitos, assim como avaliação de nossa interface.

2 Revisão bibliográfica

Neste capítulo faremos uma revisão da bibliografia existente para realizar nossa fundamentação teórica e entendimento de conceitos importantes para o desenvolvimento do trabalho, assim como detalhar projetos que utilizam programação visual para construção de novas interfaces de programação em contextos diversos. A Seção 2.1 trata sobre programação visual e alguns projetos exemplos.

2.1 Programação Visual

A programação visual tem como objetivo que os usuários possam criar, estender e personalizar seus próprios programas usando elementos gráficos bidimensionais (JOST et al., 2014).

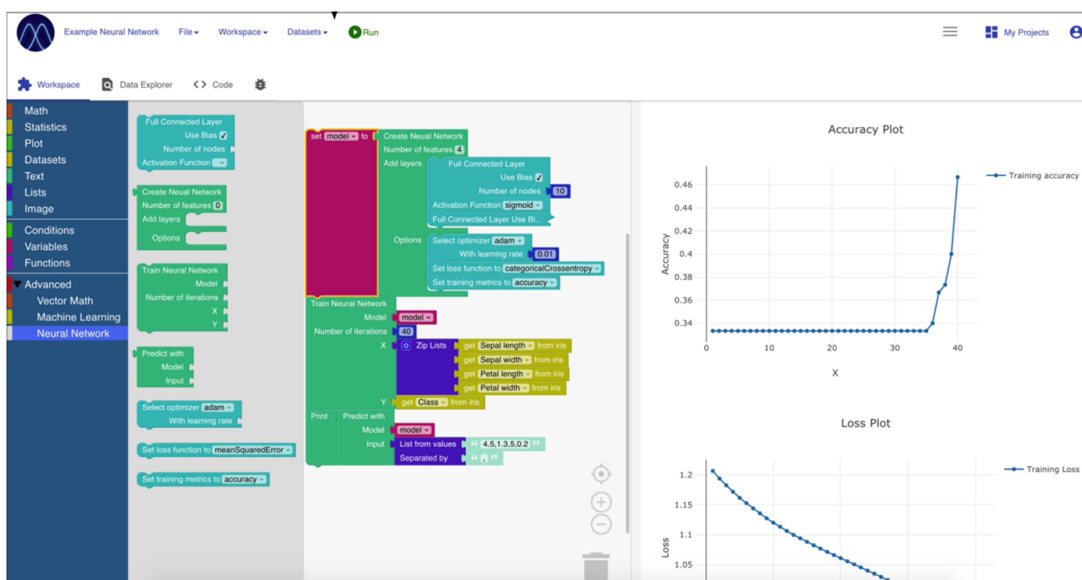
Um grande número de aplicações estão sendo desenvolvidas por usuários que não tem uma educação formal em desenvolvimento de software, e isso levou a grandes empresas como Microsoft e Amazon a investir em soluções *low-code*, que permite que usuários não-programadores desenvolvam suas aplicações. (KUHAIL et al., 2021)

O *Simulink*[1] é um software muito conhecido no meio da Engenharia, nele podemos arrastar componentes que simulam um determinado comportamento na Engenharia, por exemplo, criar um modulador e demodulador de sinal, mudar variáveis como tipo de onda, frequência, etc. Todos os blocos que constituem um determinado circuito elétrico podem ser encontrados na biblioteca de componentes e a combinação desses blocos, componentes visuais, gera uma lógica e código MatLab internamente, deixando o usuário focar em conceitos de sua área de atuação como intensidade e frequência de onda, e abstraindo conceitos técnicos da própria linguagem como ponteiros, algoritmos e estruturas de dados.

No campo da educação um exemplo é o Milo (Figura 5) (RAO; BIHANI; NAIR, 2018), um ambiente de programação visual para análise de dados, focada em um publico novato em programação ou não-programadores. A interface foi feita com base em blocos gráficos que abstraem informações e implementações complexas de cursos típicos de Análise de Dados. Conceitos como estatística básica, álgebra linear, probabilidade distributivas, algoritmos de aprendizado de máquina e mais. A implementação da interface gráfica foi feita utilizando o Blocky[2], biblioteca de geração de código a partir de elementos visuais escrita em JavaScript e desenvolvida pela Google. O Milo gera o código JavaScript equivalente de cada bloco gráfico para que o mesmo seja executado pelo próprio navegador em tempo de

[1] https://www.mathworks.com/products/simulink.html
[2] https://developers.google.com/blockly

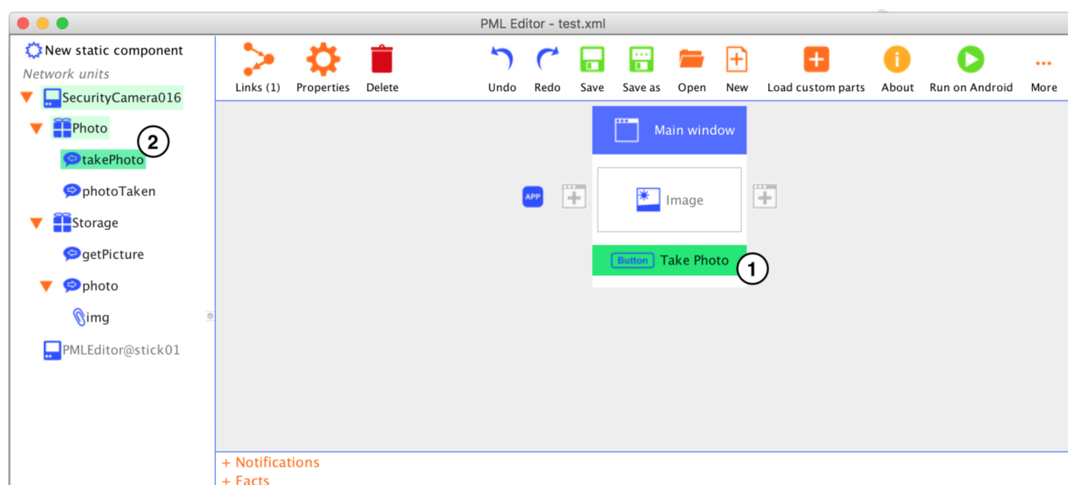
Figura 5 – Projeto Milo



Fonte: (RAO; BIHANI; NAIR, 2018)

execução, foi realizada também uma verificação de conexão entres os blocos para evitar erros sintáticos, gerando um código sintaticamente correto e sem falhas lógicas. De maneira geral o Milo cumpre bem seus objetivos e ajudou usuários em cursos introdutórios de Análise de Dados. Como crítica ao trabalho, observamos a falta de um número maior de usuários leigos não-programadores nos testes de usabilidade, aspecto elucidado no próprio texto.

Figura 6 – IoT



Fonte: (JOHNSSON; MAGNUSSON, 2020)

Para o campo da internet das coisas (do inglês, Internet of Things – IoT) foi proposta uma nova maneira de produzir interfaces gráficas (JOHNSSON; MAGNUSSON, 2020). Os usuários finais devem construir seus sistemas IoT combinando dispositivos pré-existentes em tempo real e podendo ajustar, modificar e estender o sistema criado. O projeto foi feito levando em conta que a quantidade de linhas de código escritas pelo usuário devem ser minimizadas para que sua experiência seja mais produtiva, prática conhecida como "codeless". A ideia proposta é que o usuário escolha componentes visuais (Figura 6) de um menu, arrastando para um espaço livre para especificar depois sua função e propriedades. Essa ferramenta foi então avaliada e chegou-se à conclusão que para usuários familiarizados com o Android Studio (plataforma utilizada no trabalho), a ferramenta proposta levou menos tempo para concluir os mesmos desafios de desenvolvimento comparada com forma tradicional de desenvolvimento. Fica um ponto de atenção que o estudo não avaliou cenários com usuários totalmente inexperientes, isso atrapalha a acessibilidade da ferramenta para usuários não-programadores.

Um outro projeto (THAMSEN et al., 2016) busca prover uma programação visual para a criação de fluxos de processamento envolvendo ciência de dados em um contexto de sistemas distribuídos. A ideia foi de utilizar o Flink, um framework semelhante ao Spark, para criar e distribuir trabalhos a fim de criar operações em entradas como arquivos e a partir de operações pré definidas pelo usuário criar processamento de alto desempenho gerando a saída esperada, tudo de forma visual utilizando componentes do estilo "drag and drop" funções definidas como "Map", "FlatMap"e "Reduce". Para a avaliação do projeto foi então atribuído a usuários já familiarizados com ciência de dados a tarefa de criar um programa que seria capaz de contar palavras de uma entrada arbitrária. Foi desenvolvido então uma API que processava entradas para um formato do tipo JSON, sendo então essa entrada processado pelo backend e gerando código a partir da interação do usuário com a ferramenta, tudo num espaço livre e interativo. Desenvolvido o projeto, foi então avaliado se o mesmo ajudou no desenvolvimento de novas aplicações de ciências de dados diminuindo sua curva de aprendizado e a maioria de seus usuários relatou no quesito usabilidade notas altas à ferramenta. Foi analisado também se a ferramenta ajudou no tempo de desenvolvimento, os usuários classificaram como um tempo menor para tarefas simples, porém sem grandes mudanças e talvez até limitante para tarefas mais complexas.

Em particular, observamos uma falta de propostas de programação visual para mineração de padrões em grafos. Por isso, defendemos que uma proposta de programação visual para mineração de padrões em grafos pode aumentar a atratividade da área para usuários leigos e simplificar algumas tarefas complexas para programadores não especializados (KUHAIL et al., 2021).

3 Desenvolvimento

Para o desenvolvimento completo deste trabalho, primeiro definir como funciona nossa ferramenta (Fractal seção 3.1) responsável por fazer a MPG, depois é necessário uma arquitetura (seção 3.2) do projeto que serve de guia para a construção e entendimento de funcionamento do mesmo, logo após é necessário definir a construção de nossa interface (*Frontend* seção 3.3) em si, como o usuário deve interagir com a interface, como o código fonte da ferramenta MPG é gerado e como que esse código gerado é transportado para nossa próxima camada da interface. No *Backend* (seção 3.4) foi necessário definir o comportamento de nosso servidor intermediador de sessões e também a execução da aplicação MPG, retornando então para a interface visual.

3.1 *Fractal*

Atualmente contamos com diversos algoritmos e soluções que permitem resolver problemas de MPG, e então é necessário escolher algum que nos atenda. Como o objetivo deste trabalho é desenvolver a uma interface, não faz sentido implementar uma solução de MPG do zero.

A solução escolhida é o *Fractal* (DIAS et al., 2019), um sistema de propósito geral que provê uma interface de programação projetada para facilitar o desenvolvimento de programas de MPG. Para contexto do trabalho é importante explicarmos os problemas que serão comuns ao contexto do trabalho:

- **Extração e contagem de *Motifs*.** Um *motif* ou padrão é definido como sendo um subgrafo conectado de um Grafo de entrada G. O objetivo aqui é contar a frequência que determinado *motif* aparece que contenham k vértices. Esse tipo de problema geralmente ignora os rótulos de G e é muito utilizado no ramo de bioinformática (MILO et al., 2002).
- **Listagem e contagem de Cliques.** Um k-nó clique é um subgrafo completo que contém k nós em um Grafo de entrada.

O Fractal é capaz de tratar diversos problemas relacionados a grafos, além dos que serão abordados neste trabalho. Entre eles, destacam-se o Frequent Subgraph Mining (FSM), que consiste em obter todos os subgrafos de um grafo rotulado que possuem um padrão frequente, definido por um limiar arbitrário. Outra tarefa que pode ser realizada pelo Fractal é a Listagem de subgrafos com determinado padrão, que consiste em listar todos os subgrafos isomórficos ao padrão definido pelo usuário a partir de um único grafo de entrada.

Além disso, é possível fazer Buscas baseadas em palavra-chave, utilizando rótulos atribuídos aos nós e arestas do grafo de entrada. Embora problemas de Mineração de Padrões em Grafos (MPG) sejam semelhantes a outros problemas computacionais, o contexto em que se encontram requer grande poder de processamento. Para lidar com essas operações, ferramentas específicas são necessárias, e o Fractal é uma delas, oferecendo desempenho satisfatório e uma interface de programação de fácil entendimento para a solução de todos os problemas citados. Para que se entenda de forma detalhada nossa ferramenta MPG escolhida, vamos primeiro definir qual a estrutura básica de toda aplicação construída sobre o Fractal. O *Fractal* é uma interface de programação que é focada em operações que envolvam subgrafos, portanto todas as operações que serão citadas necessitam de um objeto-estado compatível, esse objeto definiremos como sendo o *fractoid*, um objeto que representa o estado de execução do *Fractal*. Pode ser derivado de outro *fractoid* ou do Grafo de entrada.

```
1
2   import br.ufmg.cs.systems.fractal._
3   import br.ufmg.cs.systems.fractal.pattern.Pattern
4   import br.ufmg.cs.systems.fractal.util.Logging
5   import org.apache.hadoop.io.LongWritable
6   import org.apache.spark.{SparkConf, SparkContext}
7
8   object MyFractalObject extends Logging {
9     def main(args: Array[String]): Unit = {
10      // Configuracao inicial
11      val conf = new SparkConf().setAppName("MyFractalApp")
12      // Inicializacao do ambiente Spark e Fractal
13      val sc = new SparkContext(conf)
14      val fc = new FractalContext(sc)
15      // Grafo de entrada
16      val graphPath = args(0)
17      val fgraph = fc.textFile (graphPath)
18
19      // Inicio aplicacao MPG
20      // ....
21      // ....
22      // Fim aplicacao MPG
23
24      // Limpeza do ambiente Fractal e Scala
25      fc.stop()
26      sc.stop()
27    }
28  }
```

O código em questão é um exemplo de implementação de uma aplicação de Mineração de Padrões em Grafos (MPG) utilizando a biblioteca Fractal em conjunto com o ambiente de processamento distribuído Spark.

Na primeira parte do código, são importados os pacotes necessários para o funcionamento da aplicação, incluindo a biblioteca Fractal, responsável por fornecer as ferramentas de processamento de grafos.

Em seguida, a aplicação inicia a configuração do ambiente Spark com o nome "MyFractalApp", utilizando as configurações padrão. Depois, é criada uma instância do contexto Fractal a partir do contexto Spark recém-criado, que será utilizado para processar o grafo de entrada.

Na sequência, é especificado o caminho para o grafo de entrada, que é lido pelo Fractal a partir do sistema de arquivos distribuído do Spark.

Após o carregamento do grafo, a aplicação executa as operações de Mineração de Padrões em Grafos (MPG), cujo código não é especificado no trecho apresentado.

Por fim, são realizadas as operações de limpeza do ambiente Fractal e Scala, interrompendo as instâncias de contexto e encerrando a aplicação de forma adequada.

Com essa estrutura básica definida podemos exemplificar cada operador possível da nossa ferramenta, o *Fractal* contém 3 operadores primitivos, ou seja, operadores básicos que usaremos para construir todo nosso programa. São eles *extension*, *aggregation* e *filtering*. Esses operadores podem servir para enumerar subgrafos ou sumarizar os resultados.

- ***Extension (E)***. Trata-se da operação que representa a enumeração de um subgrafo com k-vértices. Recebe um conjunto de subgrafos como entrada e estende o mesmo usando sua vizinhança, produzindo outro conjunto de subgrafos. Essa extensão pode ser de 3 tipos diferentes, induzida por vértices, arestas ou por um padrão.

  Aqui temos 3 possibilidades diferentes de expansão, sendo elas: por vértices, por arestas e por padrões definidos. Em nosso contexto do trabalho será iremos focar na expansão por vértices. Podemos ver abaixo como é feita a expansão por vértices no Fractal.

```
1        // Todos os subgrafos com 3 vertices, expansao por vertice
2        val AGG_MOTIFS = "motifs"
3        // Aqui definimos que a aplicacao e do tipo VFractoid. Entao num primeiro
             momento, o Fractal ira procurar subgrafos com ao menos 1 vertice, seguindo
             ate 3. O operador expand() nesse caso, recebe apenas um argumento que se
             trata do numero de expansoes a ser executada
4        val motifs = fgraph.vfractoid.
5          expand(3)
6
7        .....
```

  Esse trecho de código define a busca por subgrafos com 3 vértices, onde a expansão é feita por vértice. A variável “AGG MOTIFS” é uma string que será utilizada como nome de uma agregação que será feita posteriormente. A aplicação é do tipo VFractoid, o que significa que o Fractal irá procurar subgrafos com pelo menos 1 vértice e seguirá até 3 vértices. O operador expand() recebe apenas um argumento que representa o número de expansões a serem executadas. A busca pelos subgrafos é armazenada na variável motifs.

- ***Aggregation (A)***. Essa primitiva tem como função sumarizar um conjunto de subgrafos em padrões para processamento posterior. Ela mapeia cada subgrafo para um conjunto de (chave, valor) que será utilizado para uma redução/sumarização posterior. Tem como entrada, uma função que extrai a chave de um subgrafo, uma segunda função que atribui valor a essa chave e uma função de redução que deve sumarizar os subgrafos de acordo com chaves semelhantes. É uma operação necessária para qualquer tipo de contagem de frequência, por exemplo.

  Vamos ver abaixo como classificar e contar subgrafos de tamanho k = 3, nesse caso o Aggregation serve para fazer a contagem e sumarização dos subgrafos existentes.

```
1      // ....
2      // Encontrando Motifs com k = 3, sendo k o número de vértices.
3      val AGG_MOTIFS = "motifs"
4      val motifs = fgraph.vfractoid.
5        expand(1).
6        // Aggregate obrigatoriamente necessita que tenha ocorrido uma expansão
           anterior, tem 4 argumentos:
7        // Arg0.: Nome da agregação (String).
8        // Arg1.: Uma função que deve definir a chave.
9        // Arg2.: Uma função que deve definir o valor associado ao par (chave, valor).
10       // Arg3.: Uma função responsável pela sumarização dos subgrafos de acordo com
           as chaves de cada.
11       aggregate [Pattern,LongWritable] (
12         AGG_MOTIFS,
13         (e,c,k) => { e.getPattern },
14         (e,c,v) => { v.set(1); v },
15         (v1,v2) => { v1.set(v1.get() + v2.get()); v1 }).
16       explore(2)
17
18     val motifsMap =
19         motifs.aggregationMap[Pattern,LongWritable](AGG_MOTIFS)
20     for ((motif,count) <- motifsMap) {
21       logInfo(s"motif=${motif} count=${count}")
22     }
```

  No trecho de código apresentado, temos a utilização do método aggregate para sumarizar os subgrafos encontrados anteriormente na operação de expansão. O método recebe quatro argumentos: o nome da agregação, uma função que define a chave, uma função que define o valor associado à chave e uma função responsável por sumarizar os subgrafos de acordo com suas chaves.

  Nesse caso, a agregação tem o nome "motifs", a chave é obtida a partir da função e.getPattern, que retorna o padrão encontrado em cada subgrafo, o valor é definido como 1 e a função de sumarização simplesmente soma os valores para cada chave.

  Após a realização da agregação, é realizada uma nova expansão com o método explore e em seguida, é criado um mapa dos padrões encontrados e suas respectivas contagens através do método aggregationMap. Por fim, um loop é utilizado para imprimir cada padrão encontrado e sua contagem no grafo de entrada.

- ***Filtering (F)***. Operador responsável por filtrar um conjunto de subgrafos que não atendam a determinado critério definido pelo usuário. Recebe uma função de entrada, que define a condição de filtragem retornado um Booleano que define se a condição foi atendida ou não.

  Ao invés de procurar motfis vamos fazer uma contagem de subgrafos do tipo Clique com k = 3, sendo k o número de vértices.

```
1      val motifs = fgraph.vfractoid.
2      expand(1).
3      filter { (e,c) => e.numEdgesAdded == e.getNumVertices - 1 }.
4      expand(1).
5      filter { (e,c) => e.numEdgesAdded == e.getNumVertices - 1 }.
6      expand(1).
7      filter { (e,c) => e.numEdgesAdded == e.getNumVertices - 1 }.
8      aggregate [Pattern,LongWritable] (
9        "motifs",
10       (e,c,k) => { e.getPattern },
11       (e,c,v) => { v.set(1); v },
12       (v1,v2) => { v1.set(v1.get() + v2.get()); v1  }
13    )
14     val motifsMap = motifs.aggregationMap[Pattern,LongWritable]("motifs")
15
16     for ((motif,count) <- motifsMap) {
17        logInfo(s"motif=${motif} count=${count}")
18     }
```

  No contexto apresentado, o operador filter está sendo usado para filtrar subgrafos que atendam a uma determinada condição. No caso, a condição é que o número de arestas adicionadas no subgrafo seja igual ao número de vértices menos 1.

  O primeiro filter é aplicado após a primeira expansão do Fractal e filtra os subgrafos que atendem à condição acima mencionada. Em seguida, a segunda expansão é aplicada e um segundo filter é usado para filtrar novamente os subgrafos que atendem à mesma condição. Finalmente, a terceira expansão é aplicada e um terceiro filter é usado para filtrar subgrafos que, mais uma vez, atendem à condição mencionada.

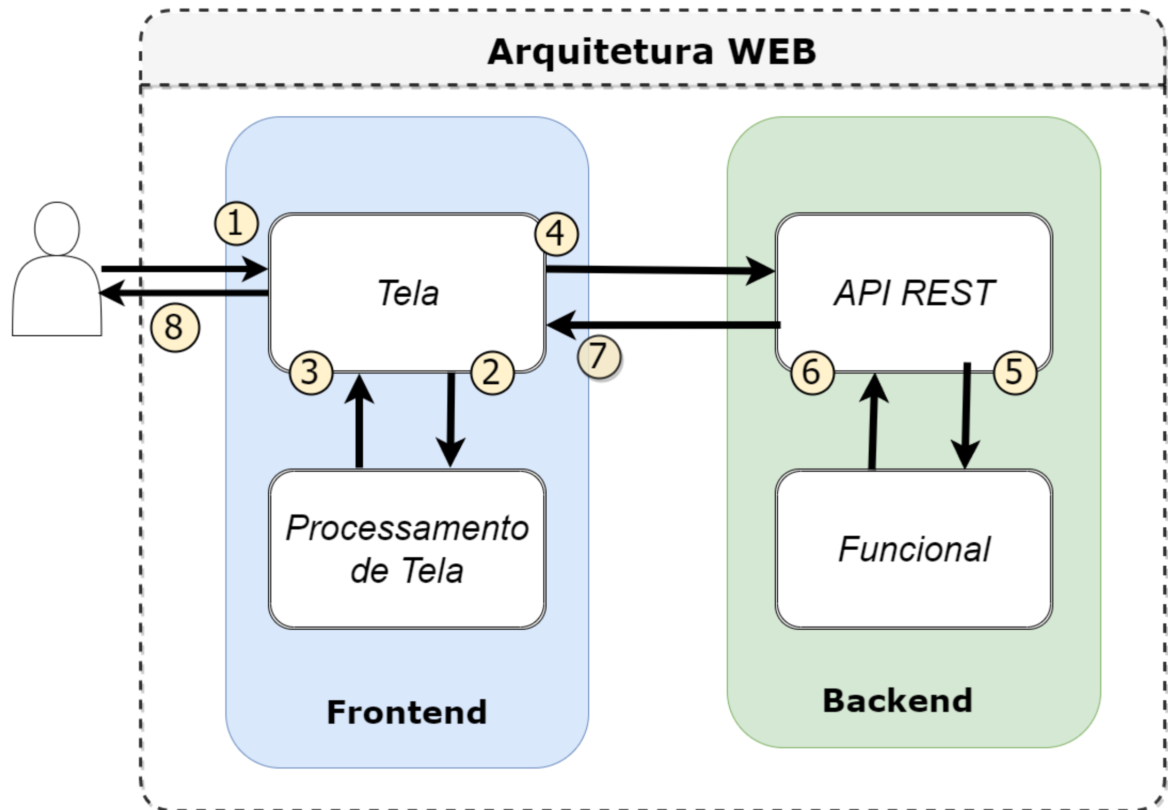
  Depois de aplicar os três filtros, o resultado é agregado usando o operador aggregate para contar o número de subgrafos que atendem à condição especificada. O nome da agregação é definido como "motifs"e a função agregadora conta o número de subgrafos usando a classe LongWritable. O resultado é então armazenado em motifsMap, que é um mapa que mapeia cada padrão de subgrafo para o número de vezes que esse padrão ocorreu nos subgrafos filtrados. Finalmente, um loop é usado para imprimir os resultados.

3.2 Arquitetura do Projeto

Na Figura 7 abaixo é possível observar como nosso projeto foi pensado, é possível observar duas partes iniciais, igualmente importantes. O *Frontend* e o *Backend*, o primeiro

tem como objetivo gerar uma página WEB interativa e que possuiu elementos visuais, já o *Backend* tem como objetivo atender as requisições HTTP vindas do *Frontend*, aqui é onde nossa aplicação executa de fato.

Figura 7 – Arquitetura do Projeto



Fonte: Produzido pelo Autor

É possível observar na Figura 7 que existem Etapas sequenciais, essas etapas ocorrem toda vez que o usuário gera um programa funcional e executa o mesmo em nossa interface de programação visual.

Etapa 1. Aqui o usuário deve selecionar o Grafo de entrada, sendo este um arquivo de texto e montar com os blocos disponíveis sua aplicação MPG.

Etapa 2. Nessa etapa enviamos todas as informações de como os blocos estão posicionados em tela, possíveis argumentos e qual foi a ação realizada pelo usuário.

Etapa 3. Nessa etapa acontece a interpretação das informações enviadas previamente. Feita a interpretação dos blocos no bloco "Processamento de Tela" temos um texto que representa o código executável que retorna para a tela.

Etapa 4. Nessa etapa o texto que representa o código gerado é enviado para o bloco "API Rest", sendo esse responsável de manter a comunicação entre a tela e a solução desenvolvida.

Etapa 5. O bloco "API Rest" é responsável por receber as requisições vindas do *Frontend*, assim como manter a sessão de cada usuário do sistema funcionando. Uma vez recebido o código que será executado e o Grafo escolhido estar disponível no servidor, é feita uma requisição WEB do tipo POST para o bloco responsável pelo funcionamento da aplicação.

Etapa 6. Aqui nessa etapa o código gerado na nossa interface executa em nosso bloco "Funcional" e então logo após seu processamento, temos um retorno que retorna a nossa "API Rest".

Etapa 7. Uma vez finalizado o processamento que o bloco "Funcional" processou, temos um retorno que será encaminhado de volta ao usuário.

Etapa 8. Uma vez com a resposta recebida, podemos exibir ao usuário quais foram os subgrafos que foram processados em nossa aplicação MPG.

3.3 *Frontend*

Nesta seção iremos descrever como se deu o desenvolvimento da parte mais externa de nossa interface, e a única que o usuário tem contato direto. O *Frontend* é composto por um *Framework* e uma biblioteca *JavaScript*.

Em vista o crescimento da área de programação e aplicações da mesma, grandes bibliotecas estão sendo desenvolvidas a fim de tornar a elaboração de uma interface de programação visual mais simples de se realizar. Uma delas é o *Blockly*[1] que conta com elementos gráficos já definidos e geração de código sintaticamente correto de forma simples de implementar e dar manutenção. O projeto apesar de ser *Open source* conta com amplo desenvolvimento e documentação por parte da *Google* o que torna menos complexo a implementação de VPLs.

O *Blocky* é a biblioteca *JavaScript* que adiciona um editor visual de código para aplicações web e aplicações móveis. O editor *Blockly* usa blocos bidimensionais para representar conceitos como varáveis, operadores, expressões lógicas, loops, condições. Esses blocos podem ser configurados para possuírem diferentes formas, encaixes, cores e comportamento.

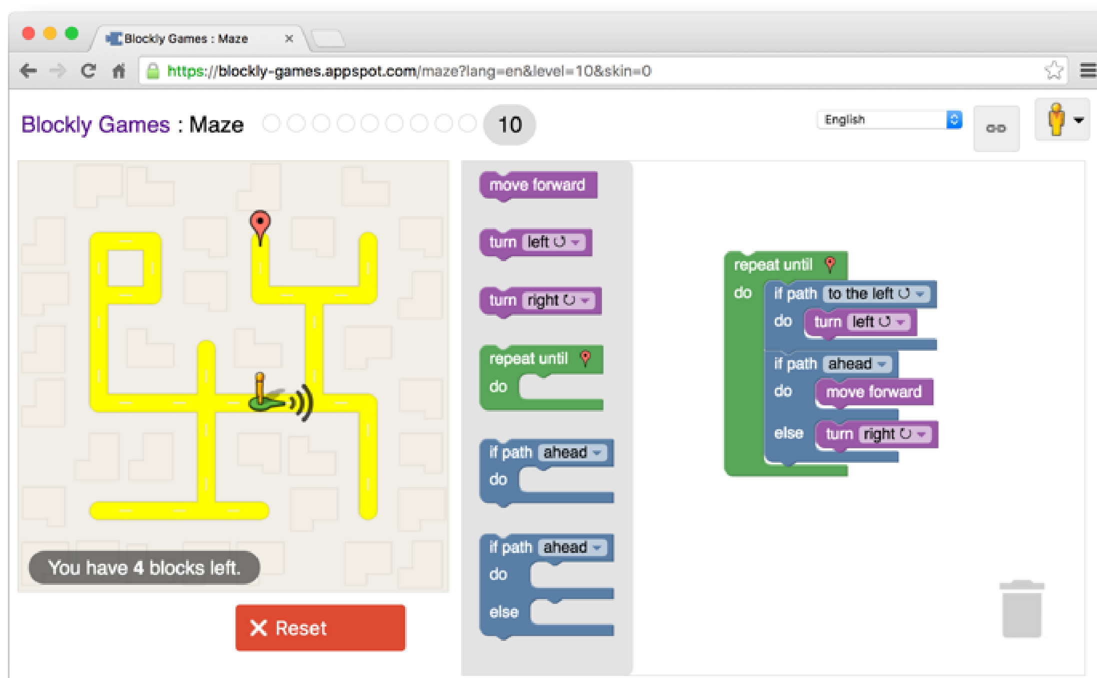
> Como uma biblioteca, *Blockly* não é nem uma linguagem completa nem ou uma aplicação pronta para usuários finais. Ela provém uma gramática e

[1] https://developers.google.com/blockly

> uma representação de programação que os desenvolvedores podem utilizar para desenvolver suas próprias aplicações. (PASTERNAK; FENICHEL; MARSHALL, 2017)

Isso significa que cada desenvolvedor deve entender a biblioteca e moldá-la de acordo com suas preferências e requisitos. Sua vantagem principal é criar uma base onde se é possível construir seus blocos personalizados sem a necessidade de definir funções de renderização, movimentação dos blocos e tarefas mais complexas como otimização.

Figura 8 – Exemplo de aplicação *Blockly*



Fonte: Blockly

Na Figura 8 podemos ver um jogo que foi construído a partir de blocos que compõem o *Blockly* e outros blocos personalizados que foram compostos pelo próprio desenvolvedor.

Além de uma biblioteca para construção de nossos blocos lógicos, é necessário também uma base de código que tem função cuidar da parte de renderização da página Web.

Para este trabalho optamos por utilizar o Vue.js, um *framework* leve e extremamente simples de se utilizar, toda página Web precisa manipular elementos da DOM, a função de qualquer *framework* é de justamente tornar mais simples e fácil esse processo.

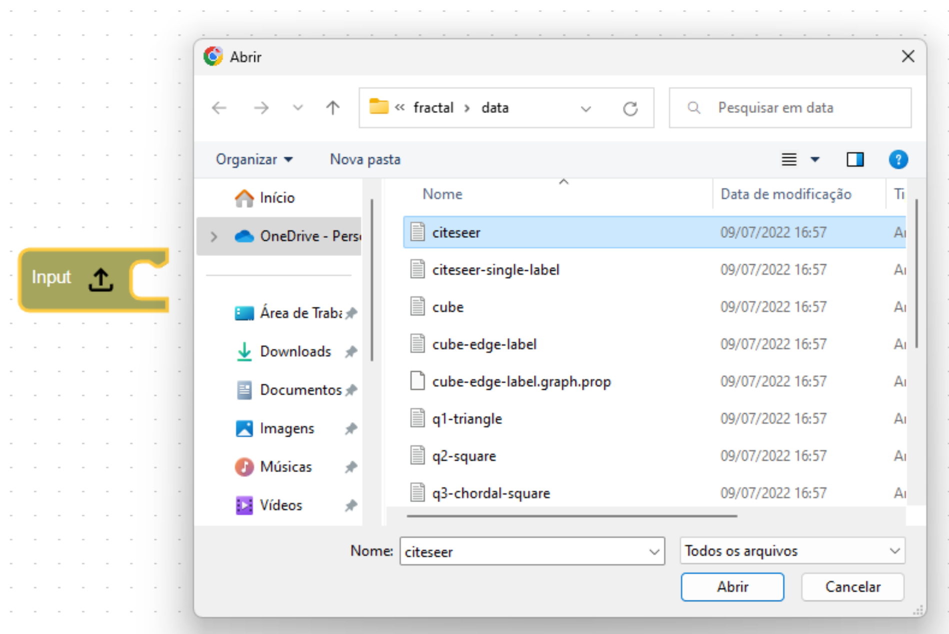
A ideia principal é modularizar os elementos em componentes únicos que podem ser reutilizados. Neste trabalho sua função principal será de cuidar todos os elementos

externos ao *Blockly*, como visual de botões, pop-up's, caixas de diálogo e reatividade da página como um todo.

3.3.1 Interação com usuário

A interface que foi desenvolvida tem elementos de interação com o usuário, por exemplo é possível que os usuários arrastem os blocos funcionais da maneira que desejar na tela, dentro de um espaço determinado. Uma interação que irá ocorrer por exemplo, é o bloco (Figura 9) responsável por fazer o *upload* do arquivo de entrada para o servidor de nossa aplicação, quando o usuário posicionar o bloco na tela, o mesmo pode receber um clique e então abrir uma janela que é responsável pelo *upload* do arquivo.

Figura 9 – Bloco de entrada



Fonte: Produzido pelo Autor

3.3.2 Geração de código

Cada bloco de nossa interface representa um comando ou função específica e os blocos podem ser conectados para criar um programa completo. Os blocos então são traduzidos para código em uma linguagem de programação específica, como JavaScript ou Python, que pode ser executado em um computador. Em nosso caso será gerado código JavaScript.

Blockly usa um conceito chamado "tradução direcionada por sintaxe" para converter os blocos em código. Isso significa que a estrutura e o arranjo dos blocos determinam

a saída final do código. Por exemplo, se um usuário arrastar um bloco "se" e um bloco "move para frente" e conectá-los, o código gerado conterá uma instrução "se" que controla o movimento de um cursor ou objeto.

Dessa forma, o Blockly permite que os usuários criem programas sem precisar conhecer a sintaxe de uma linguagem de programação específica, tornando-o uma ferramenta útil para ensinar programação para iniciantes e usuários não técnicos.

Para exemplificar vamos mostrar como cada bloco é gerado, e como cada bloco pode interpretar os argumentos fornecidos pelo usuário a fim de gerar um código executável. Definir um bloco personalizado no Blockly é um processo simples e direto que envolve a criação de uma nova classe de bloco e a registração desta classe com o Blockly. Aqui está um exemplo de como definir um bloco personalizado que exibe uma mensagem "Olá, mundo":

```
1  // Define um novo bloco customizado via JSON
2  Blockly.defineBlocksWithJsonArray([
3    {
4      type: "hello_world",
5      message0: "Ola mundo",
6      previousStatement: null,
7      nextStatement: null,
8      colour: 230,
9      tooltip: "Exibe uma mensagem de saudacao",
10     helpUrl: "",
11   },
12 ]);
13
14 // Registra o novo codigo executavel que sera executado pelo bloco
15 Blockly.JavaScript['hello_world'] = function(block) {
16   // Gera codigo executavel
17   return 'console.log('Ola, mundo!');\n';
18 };
```

Listing 3.1 – Código para um novo Bloco

Este é um exemplo de um bloco definido no Blockly. Ele não possui entrada ou saída de dados, e é usado apenas para fins de demonstração. O bloco é definido em um objeto JSON que descreve a aparência e o comportamento do bloco, incluindo a mensagem exibida, a cor do bloco, a dica de ferramenta e a URL da ajuda. Este exemplo ilustra como é fácil criar novos blocos no Blockly para personalizar a experiência de programação visual.

Também é necessário definir uma função que gere o código correspondente ao bloco. Neste exemplo, a função gera código JavaScript que exibe uma mensagem "Olá, mundo!" no console.

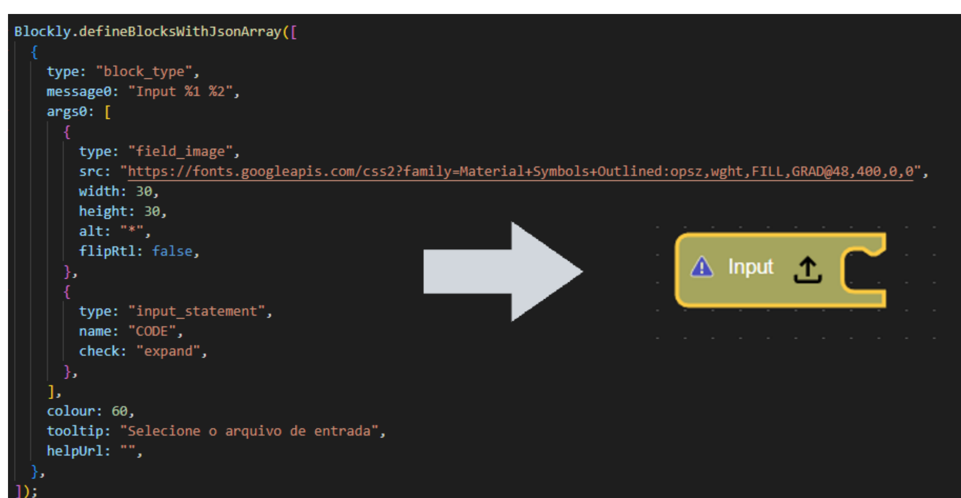
Depois de definir e registrar o bloco personalizado, você pode adicionar à sua área de trabalho do Blockly e usar imediatamente.

3.3.3 Geração de código para MPG

Agora que sabemos como podemos definir um bloco simples, vamos analisar os 4 blocos que fazem parte de nossa interface, como cada um foi construído e qual código é gerado.

Input. Vamos começar pelo bloco mais simples e descrever como o mesmo foi gerado, podemos notar na Figura 10 que o bloco tem uma funcionalidade aparente, assim como uma forma definida, rótulo e cor. Com o código abaixo definimos seu visual e seu comportamento.

Figura 10 – Input



Fonte: Produzido pelo Autor

O bloco "Input"é um bloco personalizado do Blockly que permite ao usuário selecionar um arquivo de entrada. Ele contém um campo clicável com um ícone de imagem e uma *tooltip* que instrui o usuário a selecionar um arquivo (Figura 9).

Quando o usuário clica no ícone da imagem, é exibido um seletor de arquivo que permite ao usuário escolher o arquivo de entrada.

Em resumo, o bloco "Input"simplifica a seleção de arquivos de entrada para aplicações que utilizam a biblioteca Fractal, permitindo que o usuário selecione um arquivo localmente e carregue seu conteúdo para uso posterior na aplicação.

Agora vamos definir o comportamento executável deste novo Bloco, em nossa interface o Input sempre será o primeiro "tijolo" a ser colocado, a fim de formar um Bloco executável com seu respectivo arquivo de entrada, que no caso irá representar um Grafo.

```
1 Blockly.JavaScript["input"] = function (block) {
2   let code = Blockly.JavaScript.statementToCode(block, "CODE");
3   return code;
```

```
4 };
```

Listing 3.2 – Gerador de código para Bloco Input

O código que você fornece no Blockly é utilizado para gerar um código executável em uma determinada linguagem de programação. No caso dessa definição de bloco em específico, estamos falando da linguagem JavaScript.

Ao criar um bloco no Blockly, você pode associar um código em JavaScript a ele, que será executado quando o bloco for utilizado em um programa. Nesse caso, o código definido para o bloco “input” é uma função em JavaScript que recebe um bloco como parâmetro e retorna o código executável associado a ele.

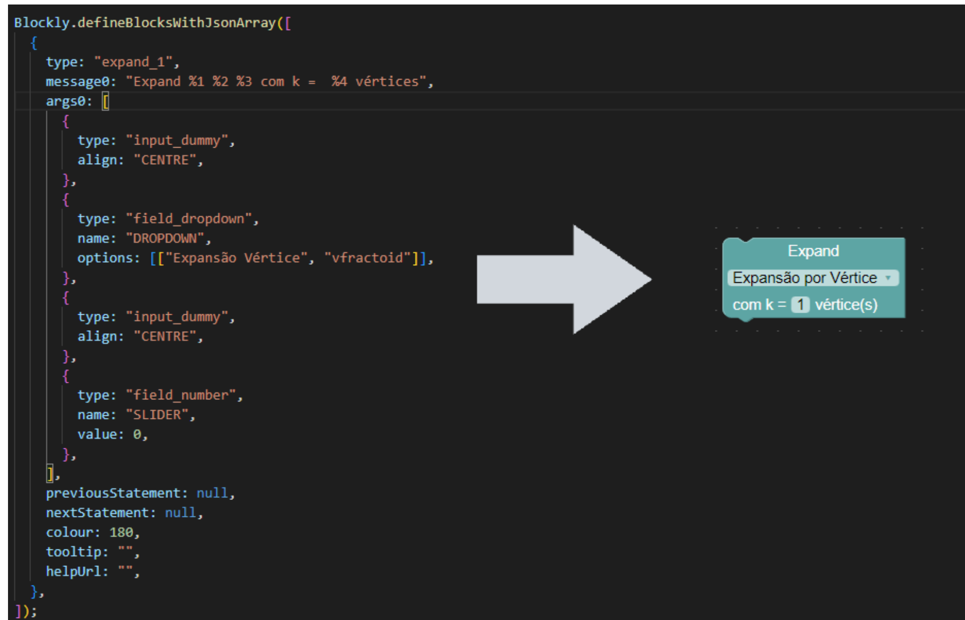
Essa função em JavaScript utiliza o método “statementToCode” para obter o código correspondente ao bloco “CODE”, que foi definido como um bloco de declaração. O método “statementToCode” é responsável por converter um bloco de declaração em código executável em JavaScript. O código gerado é então retornado pela função, que pode ser utilizado em conjunto com outros códigos gerados por outros blocos para formar um programa completo em JavaScript.

Expand. Antes é necessário recapitular o que a operação Expand representa para o Fractal. Para o Fractal a operação representa a enumeração de um subgrafo com k-vértices. Recebe um conjunto de subgrafos como entrada e estende o mesmo usando sua vizinhança, produzindo outro conjunto de subgrafos. Essa extensão pode ser de 3 tipos diferentes, induzida por vértices, arestas ou por um padrão. No contexto de nosso trabalho, vamos tratar apenas de expansões por vértices. Na figura Figura 11 podemos ver como nosso bloco que representa o operador *Expand* foi definido no *Blockly*.

O código à esquerda define um novo bloco para ser utilizado em um ambiente de programação visual baseado em blocos. O bloco é chamado de expand_1 e possui uma mensagem com quatro parâmetros:

- %: Um input_dummy, que é uma entrada que não recebe nenhum valor, e é utilizada para alinhar o bloco no centro
- %2: Um field_dropdown, que é um menu dropdown que permite selecionar uma opção a partir de uma lista de valores. Nesse caso, a opção disponível é “Expansão Vértice”, com o valor “vfractoid”.
- %3: Outro input_dummy utilizado para alinhar o bloco no centro.
- %4: Um field_number, que é um campo de entrada numérica que permite selecionar um valor inteiro. Nesse caso, o valor padrão é 0.

Figura 11 – Expand



Fonte: Produzido pelo Autor

Além dos parâmetros de mensagem, o bloco também possui outras propriedades, como a cor (180), o tooltip (uma mensagem que aparece quando o usuário passa o mouse sobre o bloco) e o helpUrl (um link para a documentação ou ajuda).

No geral, esse código é responsável por criar um novo bloco que permite ao usuário selecionar uma opção de expansão de vértice, juntamente com um valor inteiro que representa o número de vértices a serem expandidos.

```
1 Blockly.JavaScript["expand"] = function (block) {
2   let edges = block.getFieldValue("SLIDER");
3   let choice = block.getFieldValue("DROPDOWN");
4   let code = ‘.${choice}.expand(${edges}).
5         ‘;
6   return code;
7 };
```

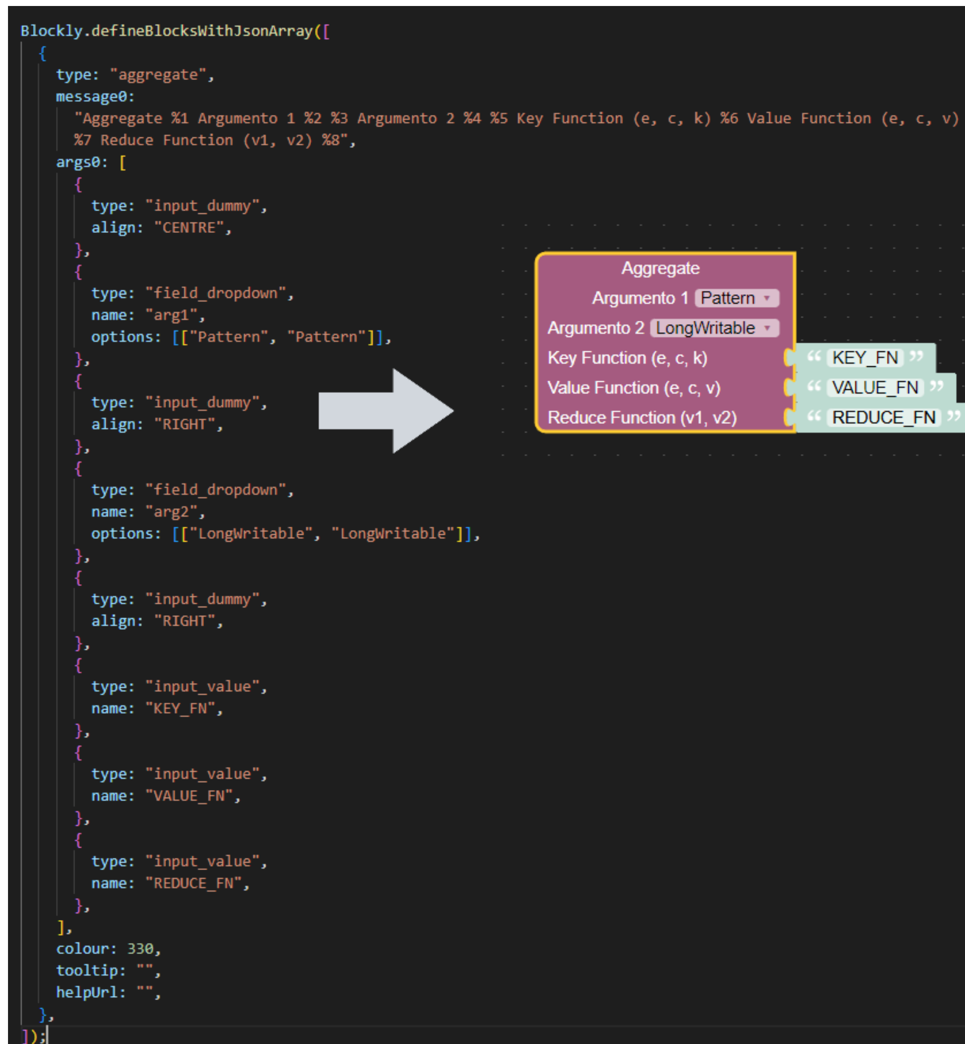
Listing 3.3 – Geração de código do Bloco Expand

O código Scala/Fractal gerado será uma string concatenada com os argumentos "choice" que representa qual tipo de expansão e "edges" que representa n-vértices da expansão.

Aggregate. Vamos relembrar a funcionalidade do Aggregate, o objetivo é sumarizar um conjunto de subgrafos em padrões para processamento posterior. Ele mapeia cada subgrafo para um conjunto de (chave, valor) que será utilizado para uma redução/sumarização posterior. Tem como entrada, uma função que extrai a chave de um subgrafo, uma segunda função que atribui valor a essa chave e uma função de redução que deve sumarizar

os subgrafos de acordo com chaves semelhantes. É uma operação necessária para qualquer tipo de contagem de frequência. Na Figura 12 definimos esse bloco visualmente.

Figura 12 – Aggregate



Fonte: Produzido pelo Autor

Esse bloco possui uma mensagem principal ("Aggregate") e cinco argumentos: dois argumentos do tipo “field_dropdown”, que permitem ao usuário selecionar uma opção de uma lista suspensa, e três argumentos do tipo “input_value”, que permitem ao usuário inserir um valor de entrada, que em nosso caso será uma string.

A mensagem principal do bloco (“Aggregate”) é definida na propriedade “message0”. Os argumentos do bloco são definidos na propriedade “args0”, em um array que contém objetos JSON que definem cada argumento.

Os dois primeiros argumentos são do tipo “field_dropdown” e permitem que o usuário selecione uma opção de uma lista. O primeiro argumento tem o nome “arg1”

e a opção [“Pattern”], enquanto o segundo argumento tem o nome “arg2” e a opção [“LongWritable”].

Os três últimos argumentos são do tipo “input_value” e permitem que o usuário insira valores de entrada. Eles têm os nomes “KEY_FN”, “VALUE_FN” e “REDUCE_FN”.

O bloco também possui uma cor (330), um tooltip e uma helpUrl, que podem ser preenchidos com informações adicionais sobre como usar o bloco.

Considerando a definição desse bloco, agora temos o código que será gerado a partir desse bloco e seus argumentos:

```
1  Blockly.JavaScript["aggregate"] = function (block) {
2    let key_fn = Blockly.JavaScript.valueToCode(
3      block,
4      "KEY_FN",
5      Blockly.JavaScript.ORDER_ATOMIC
6    )
7
8    let value_fn = Blockly.JavaScript.valueToCode(
9      block,
10     "VALUE_FN",
11     Blockly.JavaScript.ORDER_ATOMIC
12   )
13
14   let reduce_fn = Blockly.JavaScript.valueToCode(
15     block,
16     "REDUCE_FN",
17     Blockly.JavaScript.ORDER_ATOMIC
18   )
19
20   let arg1 = block.getFieldValue("arg1")
21   let arg2 = block.getFieldValue("arg2")
22   let code = ‘aggregate [${arg1},${arg2}] (
23                     "motifs",
24                     (e,c,k) => { ${key_fn} },
25                     (e,c,v) => { ${value_fn} },
26                     (v1,v2) => { ${reduce_fn}   }
27                 )
28                 val motifsMap = motifs.aggregationMap[${arg1},${arg2}]("motifs")
29         ‘;
30   return code;
31 };
```

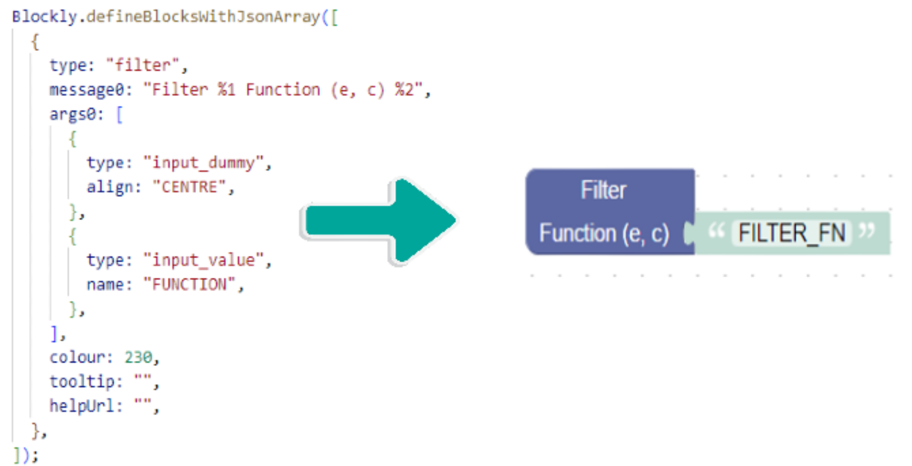
Listing 3.4 – Geração de código do Bloco Aggregate

Da linha 2 a 18 recuperamos cada String contida em cada Input do Bloco Aggregate, nas linhas 20 e 21 recuperamos o valor do primeiro e segundo dropdown respectivamente. Por fim concatenamos todos esses argumentos com o modelo “aggregate“ no Fractal. Assim como também é gerado o código que representa o “motifsMap“, sendo este o nosso resultado da agregação feita.

Filter. O último bloco de nossa interface é o Filter, responsável por filtrar um conjunto de subgrafos que não atendam a determinado critério definido pelo usuário. Recebe uma função de entrada, que define a condição de filtragem retornado um Booleano

que define se a condição foi atendida ou não. O bloco foi definido conforme a Figura 13.

Figura 13 – Filter



Fonte: Produzido pelo Autor

O bloco definido aqui é chamado “Filter” e permite aplicar uma função de filtragem que será utilizada pelo nosso operador Fractal. O campo de função é o único campo editável do bloco e aparece na forma de um campo de entrada de valor, permitindo que o usuário insira a função de filtragem.

Existem dois campos definidos no bloco: um “input_dummy”, que é utilizado apenas para alinhar os campos do bloco, e um “input_value” chamado “FUNCTION”, que é onde o usuário insere a função de filtragem.

Com essa definição feita, podemos ver agora como o código é gerado usando os argumentos do bloco.

```
1 Blockly.JavaScript["filter"] = function (block) {
2   let value_option1 = Blockly.JavaScript.valueToCode(
3     block,
4     "FUNCTION",
5     Blockly.JavaScript.ORDER_ATOMIC
6   )
7   let code = ‘filter { (e,c) => ${value_option1} }.‘;
8   return code;
9 };
```

Listing 3.5 – Geração de código do Bloco Filter

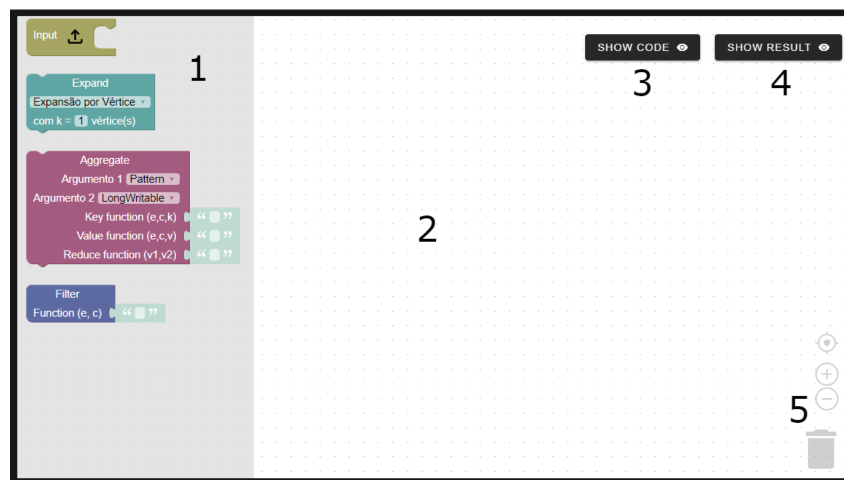
Aqui a string que nosso bloco retorna é uma concatenção da função inserida pelo usário da seguinte forma: “filter { (e,c) => $value_option1 }”.

Nesta seção vimos como cada bloco que constituem nossa interface foi construído visualmente e funcionalmente, é possível notar que temos um padrão na construção que pode ser facilmente replicado para novos blocos personalizados.

3.3.4 Geração de código em etapas

Com as definições de como cada bloco foi projetado e funciona, vamos agora a um exemplo de uma aplicação MPG funcional (Mais exemplos, consultar Figura 28 a Figura 33), partindo de um grafo G de entrada e que queremos encontrar subgrafos do tipo Motifs, no caso Motifs de tamanho, k = 3. Um *Motif* ou padrão é definido como sendo um subgrafo conectado de um Grafo de entrada G. O objetivo aqui é contar a frequência que determinado *Motif* aparece que contenham *k* vértices, ou seja, faremos uma contagem.

Figura 14 – Tela Inicial



Fonte: Produzido pelo Autor

Na Figura 14 vemos como nossa interface é, assim como as ações possíves que o usuário pode realizar quando utilizar. Em resumo podemos dizer:

- 1. Aqui temos os blocos disponíveis para utilização, basta que o usuário arraste o componente para o campo central.
- 2. Se trata do *canvas*, onde é possível mover livremente os blocos, assim como redimensionar zoom, ou fixar uma posição.
- 3. O botão "Show Code" serve para processar os blocos que foram posiconados no canvas, quando o usuário clicar no mesmo, o código equivalente é mostrado em sua tela assim como os argumentos que foram inseridos, e uma vez processado, esse código vai para o Backend(seção 3.4) que faz o processamento do código exibido na tela.
- 4. Uma vez que os subgrafos foram processados, podemos obter uma representação visual dos subgrafos que foram processados em nossa aplicação.
- 5. Ferramentas do *canvas*, como excluir blocos, zoom e fixação de tela.

Com nossa interface conhecida, agora podemos montar a aplicação MPG que fará o processamento de subgrafos do tipo Motifs com k = 3, sendo k o número de vértices. A primeira etapa é selecionar o bloco "Input", bloco este que sempre servirá de base para nossas aplicações pois o mesmo se traduz na parte fixa de nosso código.

Figura 15 – Bloco input

Fonte: Produzido pelo Autor

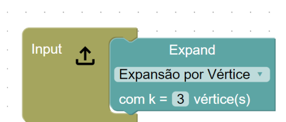
```
1  import br.ufmg.cs.systems.fractal._;
2  import br.ufmg.cs.systems.fractal.pattern.
       Pattern
3  import br.ufmg.cs.systems.fractal.util.Logging
4  import org.apache.hadoop.io.LongWritable
5
6  val fc = new FractalContext(sc)
7  val graphPath = "GRAPH_PATH";
8  val fgraph = fc.textFile (graphPath)
9  val motifsMap =
10     fgraph.
11     vfractoid
12 ////
13 /// APLICACAO MPG
14 ///
15 for ((motif,count) <- motifsMap) {
16     logInfo(s"motif=${motif} count=${count}")
17 }
18
19 fc.stop()
```

Listing 3.6 – Código equivalente

É possível observar que para a Figura 17 temos o código equivalente que a interface gera automaticamente, no caso o único argumento desse bloco é o próprio arquivo que será utilizado como entrada, lembrando que este arquivo deve representar um grafo.

Com a entrada definida, agora o próximo passo é fazer a nossa expansão, como estamos buscando subgrafos com k = 3, sendo k o número de vértices.

Figura 16 – Bloco input + expand



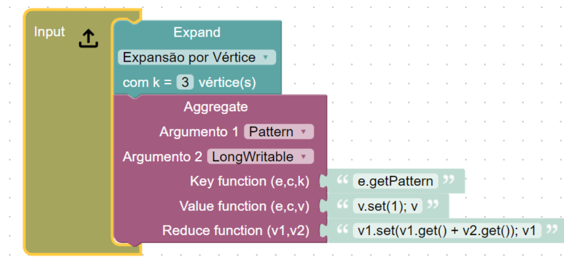
Fonte: Produzido pelo Autor

```
1  import br.ufmg.cs.systems.fractal._;
2  import br.ufmg.cs.systems.fractal.pattern.
       Pattern
3  import br.ufmg.cs.systems.fractal.util.Logging
4  import org.apache.hadoop.io.LongWritable
5
6  val fc = new FractalContext(sc)
7  val graphPath = "GRAPH_PATH";
8  val fgraph = fc.textFile (graphPath)
9  val motifs =
10     fgraph.
11     vfractoid.
12     expand(3)
13
14 for ((motif,count) <- motifsMap) {
15     logInfo(s"motif=${motif} count=${count}")
16 }
17
18 fc.stop()
```

Listing 3.7 – Código equivalente

Com as expansões feitas, é necessário agora agregar nossos subgrafos para realizar uma contagem de quantos subgrafos existem para cada padrão com k = 3 vértices. Para isso é necessário o bloco Aggregate.

Figura 17 – Motifs, k = 3



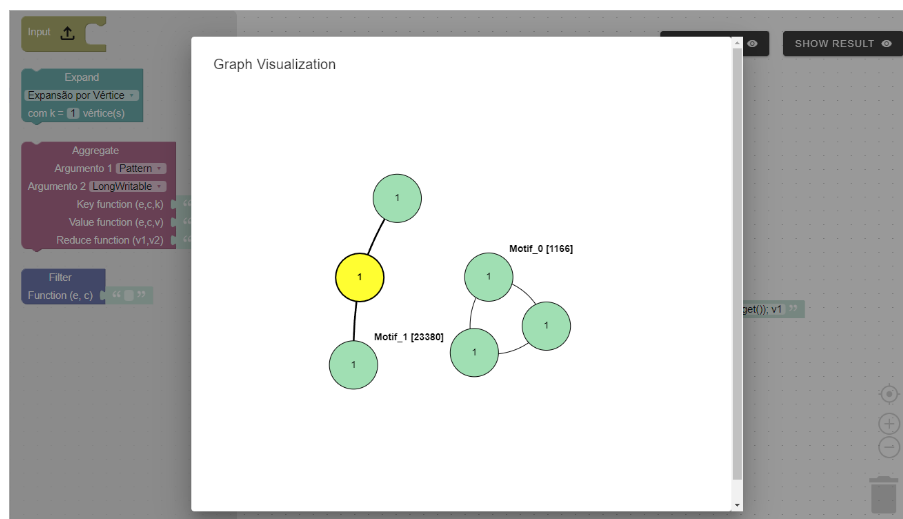
Fonte: Produzido pelo Autor

```scala
import br.ufmg.cs.systems.fractal._;
import br.ufmg.cs.systems.fractal.pattern.
    Pattern
import br.ufmg.cs.systems.fractal.util.Logging
import org.apache.hadoop.io.LongWritable

val fc = new FractalContext(sc)
val graphPath = "GRAPH_PATH";
val fgraph = fc.textFile (graphPath)
val motifs =
    fgraph.
    vfractoid.
    expand(3).
    aggregate [Pattern,LongWritable]
   ( "motifs",
   (e,c,k) => { e.getPattern },
   (e,c,v) => { v.set(1); v },
   (v1,v2) => { v1.set(v1.get() + v2.get()); v1
        } )

val motifsMap = motifs.aggregationMap[Pattern,
    LongWritable]("motifs")

for ((motif,count) <- motifsMap) {
    logInfo(s"motif=${motif} count=${count}")
}

fc.stop()
```

Listing 3.8 – Código equivalente

Esse conjunto de blocos e parâmetros já é uma aplicação do tipo MPG que processa e faz uma contagem de subgrafos com k = 3, diferenciando pelo seu padrão de vértices + arestas. Uma vez montado podemos processá-lo, e após o processamento temos um conjunto de subgrafos conforme a Figura 18, lembrando que para esse processamento foi utilizado um grafo que todos os vértices tinham o mesmo rótulo (cor). Podemos notar o padrão processado e a respectiva contagem do mesmo.

Figura 18 – Subgrafos resultantes

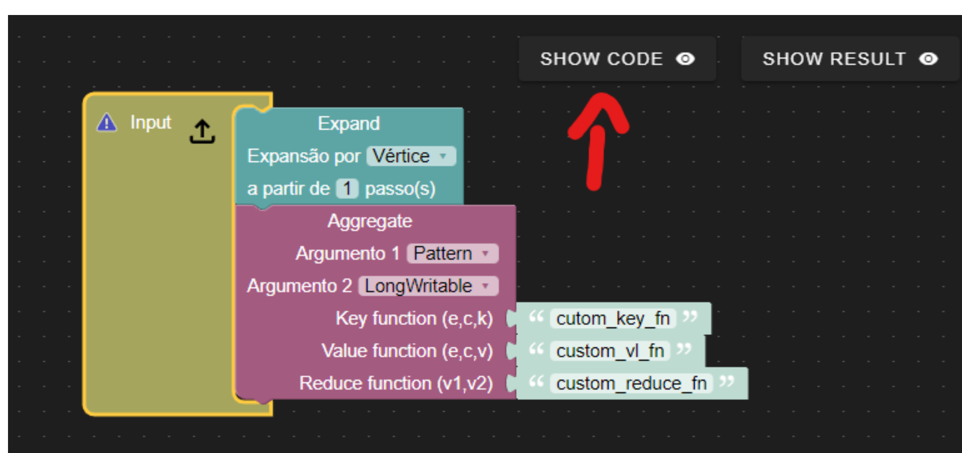


Fonte: Produzido pelo Autor

3.3.5 Comunicação com o *Backend*

Vamos considerar que o usuário utilizou a interface para gerar seu código (Figura 19), no caso uma string. O processo que transporta essa string para nosso *Backend* é o que conhecemos como comunicação por meio de requisições web. A ferramenta que será nesta etapa será o "Axios", uma biblioteca JavaScript que facilita a realização de requisições HTTP (como GET, POST, PUT, DELETE, entre outras) no lado do cliente. É baseado em promessas e fornece uma API fácil de usar para realizar requisições HTTP e lidar com respostas.

Figura 19 – Geração de código



Fonte: Produzido pelo Autor

No trecho a seguir podemos ver o que acontece quando o usuário aperta o botão responsável por gerar o código e enviar para execução.

```
1  showCode() {
2        this.outputObjects = [];
3        const workspace = this.$refs["ref_blk"].workspace;
4        this.code = BlocklyJS.workspaceToCode(workspace);
5        let match = this.code.match(".*fractoid.");
6        this.code = this.code.replace(/.*fractoid./g, "");
7        let temp_header = this.header + match[0];
8        this.code = temp_header + this.code;
9        this.code = this.code + this.footer;
10
11       axios
12         .post("http://localhost:3080/fractal/runcode", {
13           stmt: `${this.code}`,
14         })
15         .then((response) => {
16           console.log(response.data);
17           let outputString =
18             response.data.data["text/plain"].match(/output{.*}/g);
19
20           outputString.map((string) => {
21             const obj = JSON.parse(string.split(/output/g)[1]);
```

```
22            this.outputObjects.push(obj);
23          });
24          console.log(this.outputObjects);
25        })
26        .catch((error) => {
27          console.error(error);
28        });
29    },
```

Listing 3.9 – Método Show Code

Das linhas 2 a 9 temos a recuperação da String gerada pelo Blockly, assim como sua formatação final. Já da linha 11 em diante temos o processo responsável por transportar essa String que representa um código Scala/Fractal e envia-lá ao *Backend*. Neste exemplo, o método POST do Axios é usado para realizar uma requisição POST para a API, o código vai como um objeto JSON como podemos ver na linha 13. A resposta da requisição é então manipulada na promise then, onde você pode acessar os dados retornados pela API usando response.data. Em caso de erro, você pode manipular o erro na promise catch.

Além disso, o Axios também suporta requisições GET, PUT, DELETE e outras, bem como o envio de parâmetros na requisição, a configuração de cabeçalhos, a autenticação e muito mais. Em geral, o Axios é uma ferramenta poderosa e fácil de usar para realizar requisições HTTP no lado do cliente.

3.4 *Backend*

Nesta seção vamos explicar como funciona e como foi desenvolvido o *Backend* de nossa interface WEB. O *Backend* é o termo usado para se referir ao lado "invisível" de uma aplicação web, responsável por lidar com tarefas como armazenamento de dados, processamento de informações e geração de respostas para requisições feitas pelo *frontend*.

O *backend* é composto por servidores, bancos de dados e outros componentes de software que trabalham juntos para garantir que a aplicação funcione corretamente e forneça aos usuários as informações e recursos que eles precisam.

Quando um usuário acessa uma página web, ele está interagindo com o *frontend* da aplicação, que é construído com tecnologias como HTML, CSS e JavaScript. Quando o usuário faz uma ação, como enviar uma requisição de formulário ou clicar em um botão, o *frontend* envia uma requisição para o *backend*. Esse então, por sua vez, processa a requisição, busca informações do banco de dados se necessário, e retorna uma resposta para o *frontend*, que a exibe para o usuário.

Em resumo, o *backend* é responsável por garantir que a aplicação funcione corretamente e forneça informações precisas e atualizadas aos usuários. É uma parte crucial da arquitetura de uma aplicação web e pode ser desenvolvido usando uma ampla variedade

de tecnologias e linguagens de programação, como PHP, Ruby on Rails, Node.js, Java, entre outras. Em nosso caso iremos utilizar o *Node.js*.

3.4.1 Servidor de REST API

A primeira camada de todo *Backend* é o que chamamos de REST API, que é um padrão de desenvolvimento de software para criar serviços web que forneçam recursos através da Internet. É usado para permitir que aplicações se comuniquem uns com os outros e troquem informações em formato de dados estruturados, como JSON ou XML.

A REST API segue o princípio de arquitetura REST (Representational State Transfer), que é baseado em verbos HTTP, como GET, POST, PUT e DELETE, para realizar operações com recursos. Por exemplo, uma requisição GET pode ser usada para ler informações de um recurso, enquanto uma requisição POST pode ser usada para criar um novo recurso.

As REST APIs são amplamente utilizadas porque são fáceis de entender e de implementar, e porque podem ser acessadas por uma ampla variedade de dispositivos e tecnologias, incluindo smartphones, laptops e computadores.

Além disso, as REST APIs também permitem a integração fácil com outras aplicações e sistemas, o que significa que os desenvolvedores podem construir aplicações mais sofisticadas e completas usando dados e recursos de fontes externas.

Em resumo, as REST APIs são uma parte crucial da infraestrutura da web moderna e são amplamente utilizadas para criar aplicações que podem ser acessadas remotamente por dispositivos e tecnologias diferentes. Para a construção dessa nossa REST API o Node.js foi o framework utilizado.

Para exemplificar vamos demonstrar o que acontece quando o usuário faz uma requisição do tipo POST no *frontend*. O código abaixo representa o método responsável por fazer o upload dos arquvios recebidos no servidor.

```
1  router.post("/upload", async (req, res) => {
2    try {
3      if (!req.files) {
4        res.send({
5          status: false,
6          message: "No file uploaded",
7        });
8      } else {
9        let file = req.files.file;
10
11       //Use the mv() method to place the file in upload directory (i.e. "uploads")
12       file.mv("./public/" + file.name);
13
14       //send response
15       res.send({
16         status: true,
17         message: ‘File "${file.name}" was uploaded‘,
```

```
18         data: {
19           name: file.name,
20           mimetype: file.mimetype,
21           size: file.size,
22         },
23       });
24     }
25   } catch (err) {
26     res.status(500).send(err);
27   }
28 });
```

Listing 3.10 – Upload de arquivo

Vemos então que quando o usuário faz uma requisição do tipo POST na url “.../upload“ o método acima é acionado, caso não exista arquivo válido o servidor retorna para a interface uma mensagem de erro, caso o arquivo seja válido o mesmo será transferido para o servidor como podemos ver nas linhas 9 a 24, assim como uma mensagem de sucesso é enviada. Podemos notar que a API REST então é a ponte de comunicação entre *Frontend* e *Backend*, sendo responsável por processar requisições e devolver as respostas adequadas.

3.4.2 Execução de aplicações de MPG

Uma vez que temos nosso código gerado, esse código chega no servidor via a API REST, agora basta executarmos o código em um ambiente adequado. Entra então a última peça de nosso sistema WEB, o “Apache Livy“, que é a ferramenta responsável por executar nosso código de MPG.

O Apache Livy é uma biblioteca de interface de programação de aplicativos (API) distribuída que permite que os usuários interajam com um *cluster* Apache Spark de forma remota e segura. É projetado para tornar a interação com Spark mais fácil, especialmente para aplicativos que não rodam no mesmo *cluster* que o Spark. Temos os seguintes passos de como o Apache Livy funciona:

- Submissão de tarefas: O usuário envia uma tarefa para o Livy usando a API REST. A tarefa pode ser uma sessão Spark interativa, um script Spark ou uma aplicação Spark.
- Gerenciamento de sessão: O Livy gerencia a criação e o fechamento de sessões Spark em segundo plano. As sessões são agrupadas em um cluster Spark e compartilham recursos, como memória e CPU, para melhorar a eficiência.
- Execução de tarefas: O Livy envia a tarefa para o cluster Spark para ser executada. O Spark roda a tarefa e retorna os resultados para o Livy.
- Retorno de resultados: O Livy retorna os resultados da tarefa para o usuário. A resposta pode ser uma saída de texto, um gráfico ou outro tipo de resultado visual.

O Apache Livy é uma ferramenta útil para aplicativos que precisam interagir com o Spark de forma remota ou segura, mas não precisam gerenciar o cluster Spark diretamente. Além disso, o Livy é escalável e pode ser usado em ambientes em nuvem ou empresariais para aumentar a eficiência e a produtividade dos usuários.

Podemos notar abaixo como nossa API REST recebe o código gerado pelo *Frontend* e encaminha uma request para a API do Livy:

```
1  const express = require("express");
2  const LivyClient = require("livy-client");
3
4  const router = express.Router();
5
6  router.post("/runcode", async (req, res) => {
7    start(req, res);
8  });
9
10 const start = async (req, res) => {
11   const stmt = req.body.stmt;
12   // Create client
13   const livy = new LivyClient({
14     host: "localhost",
15     port: "8998",
16   });
17
18   // Get sessions
19   const sessions = await livy.sessions();
20   for (session of sessions) {
21     const status = await session.status();
22     console.log(`Session id: ${status.id}, state: ${status.state}`);
23   }
24
25   // Create session
26   const newSession = await livy.createSession({
27     kind: "spark",
28     name: "fractal_client",
29     jars: [
30       "/home/unix/libs_tcc/fractal/fractal-core/build/libs/fractal-core-SPARK-2.2.0.jar",
31       "/home/unix/libs_tcc/spark/jars/scala-library-2.11.8.jar",
32     ],
33   });
34
35   // Listen event of a session
36   newSession
37     .on("starting", (status) => {
38       console.log(
39         "Session starting... " +
40           status.log.slice(0, -1).slice(-1)[0].replace(/\n/g, " ")
41       );
42     })
43     // Once ready, execute a code and kill the session
44     .once("idle", async (status) => {
45       const statement = await newSession.run({ code: stmt });
46       statement
47         .on("running", (status) => {
48           console.log(
49             `Statement running... ${Math.round(status.progress * 100)}/100%`
50           );
```

```
51          })
52          .once("available", (response) => {
53            console.log('Statement completed. Result: ');
54            console.log(response.output);
55            newSession.kill();
56            res.send(response.output);
57          });
58      });
59  };
```

Listing 3.11 – Execução código gerado

Na linha 6 vemos que quando o *Frontend* envia uma requisição do tipo POST com a url do tipo "…/runcode", a função start() é acionada. Das linhas 26 a 33 vemos como é criada a sessão Livy com os parâmetros necessários para nossa aplicação rodar, assim como dependências necessárias. Após inciada a sessão (Linha 36), quando a sessão estiver num estado de espera (Linha 44) invocamos o método "run()" de nossa API do Livy passando nosso código gerado como argumento, após o processamento enviamos de volta ao *Frontend* o processamento do código (Linha 52).

4 Resultados

Na seção de resultados, apresentaremos as principais descobertas obtidas a partir da avaliação da interface desenvolvida. Em particular, analisaremos como a interface pode resolver o caso envolvendo os grafos Citeseer, Mico (ELSEIDY et al., 2014) e Facebook (ROZEMBERCZKI; ALLEN; SARKAR, 2019).

Todos os exemplos e passos que serão exibidos nessa seção podem ser reproduzidos utilizando os *Datasets* e a interface disponível no repositório do trabalho (RODIANI, 2023).

Na seção 4.1 faremos uma avaliação e discussão sobre como a interface ajudou a melhorar a eficiência do desenvolvimento de aplicações MPG em cada um desses grafos, destacando as principais vantagens oferecidas pela interface em relação a abordagens tradicionais de MPG.

Já na seção 4.2 avaliaremos o custo adicional que a interface teve sobre o resultado final de processamento. Analisaremos o impacto que o uso da interface teve sobre a o tempo de execução de aplicações comuns a problemas que envolvem MPG.

Por fim, apresentaremos os resultados de forma clara e objetiva, usando gráficos, tabelas e outras visualizações sempre que possível para tornar as informações mais acessíveis e fáceis de entender. Em nossas análises utilizaremos os Grafos abaixo (Tabela 1).

Tabela 1 – Grafos utilizados

Nome	**Nº de vértices**	**Nº de arestas**	**Quantidade de rótulos diferentes**	**Significado**
Citeseer	3,312	4,732	Múltiplos possíveis	Vértices: Publicações acadêmicas; Arestas: Citações entre publicações; Rótulos: Áreas de pesquisa em Ciência da Computação;
Mico	100,000	1,080,298	Múltiplos possíveis	Vértices: Autores da Microsoft Research; Arestas: Indica dois autores sendo co-autores em um artigo; Rótulos: Áreas de interesse de pesquisa desses autores;
Facebook	22,470	171,002	Múltiplos possíveis	Vértices: Páginas verificadas do Facebook; Arestas: “Likes” mútuos entre duas páginas; Rótulos: A categoria das páginas (companhia, política, etc);

4.1 Estudo de casos

Nesta seção faremos uma análise de como a interface de programação visual desenvolvida contribuiu para melhorar a eficiência do desenvolvimento de aplicações de

MPG em três grafos diferentes: Citeseer, Mico e Facebook.

Para cada um desses casos, avaliaremos como a interface tornou o processo de desenvolvimento mais intuitivo e simples para usuários não-programadores, permitindo a criação de programas de forma mais rápida e eficiente.

Serão destacadas as principais vantagens oferecidas pela interface em relação a abordagens tradicionais de MPG, como a redução da curva de aprendizado, a facilidade de visualização e manipulação dos dados, e a agilidade na implementação de algoritmos.

Além disso, serão apresentados os resultados obtidos a partir da aplicação de cada algoritmo em cada grafo, avaliando-se a eficiência da interface em comparação com abordagens tradicionais.

Com essa análise, esperamos demonstrar como a interface de programação visual desenvolvida pode ser uma solução eficiente para o problema de usabilidade e curva de aprendizado em ferramentas de processamento de grandes quantidades de dados, ampliando o leque de usuários capazes de utilizar essas ferramentas e promovendo uma democratização do acesso a tecnologias de análise de dados.

4.1.1 Citeseer

A rede Citeseer é uma rede de citações acadêmicas em Ciência da Computação, na qual cada vértice representa uma publicação e as arestas representam as citações diretas entre essas publicações. Através dessa rede, é possível analisar a distribuição e a conexão entre as áreas de pesquisa na Ciência da Computação, identificando padrões de colaboração entre pesquisadores e áreas de pesquisa relacionadas.

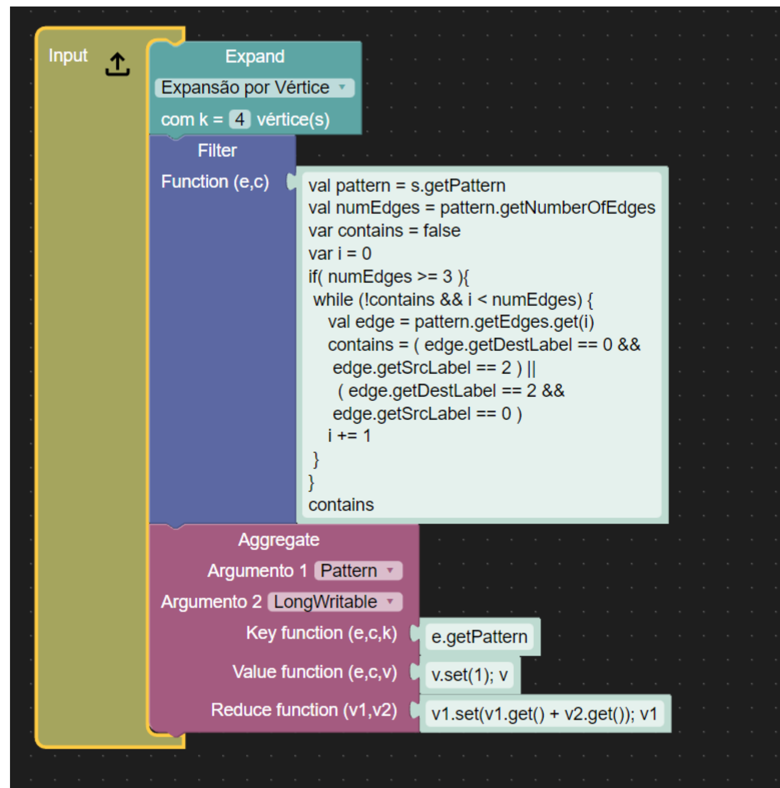
Para esse fim, podemos utilizar algoritmos de busca e mineração de dados para identificar subgrafos que apresentem características específicas, como a presença de rótulos (áreas de pesquisa) relacionadas e a presença de conexões fortes entre essas áreas.

Suponha que nosso objetivo seja criar um filtro para identificar subgrafos na rede Citeseer que contenham no mínimo 4 vértices e onde ocorra pelo menos uma aresta entre os rótulos 0 e 2, de forma que essa conexão entre os labels represente uma conexão forte. Para isso, poderíamos utilizar um algoritmo (Figura 20) de busca que percorra todos os subgrafos da rede e verifique se eles satisfazem essas condições.

Uma vez identificados os subgrafos que atendem aos critérios estabelecidos, poderíamos utilizá-los para identificar áreas de pesquisa que apresentam maior conexão e colaboração entre si. Por exemplo, poderíamos analisar a distribuição dos labels nos subgrafos identificados e identificar quais áreas de pesquisa aparecem com maior frequência em conjunto.

Com essa informação obtida na Figura 21, poderíamos incentivar a colaboração

Figura 20 – Solução montada na Interface

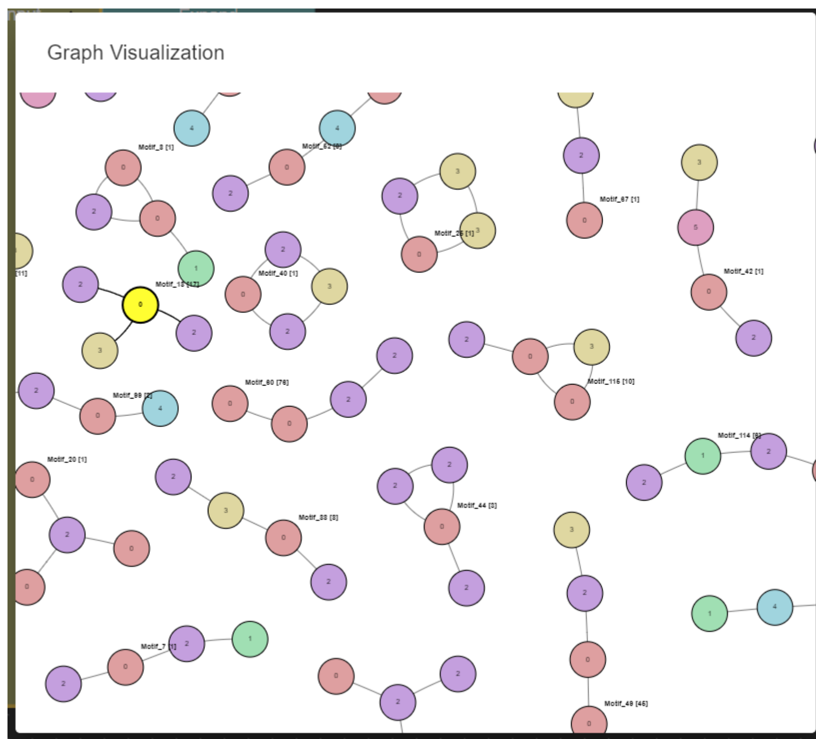


Fonte: Produzido pelo Autor

entre pesquisadores das áreas identificadas como frequentemente colaborativas, por meio da criação de programas de fomento para a realização de pesquisas conjuntas entre pesquisadores das áreas identificadas.

Em resumo, ao criar um filtro para identificar subgrafos na rede Citeseer que contenham no mínimo 4 vértices e uma conexão forte entre os labels 0 e 2, poderíamos utilizar essa informação para incentivar a colaboração entre pesquisadores de áreas de pesquisa relacionadas e fomentar a produção de pesquisas de maior impacto e relevância na área de Ciência da Computação.

Figura 21 – Uma amostra dos subgrafos resultantes



Fonte: Produzido pelo Autor

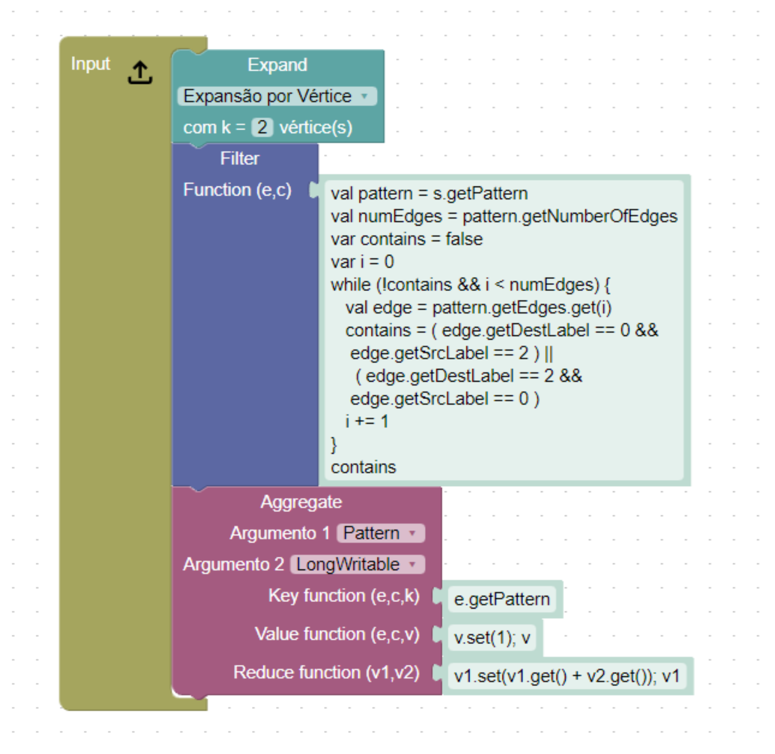
4.1.2 Facebook

O grafo do Facebook (ROZEMBERCZKI; ALLEN; SARKAR, 2019) em questão tem como elementos os vértices, que representam páginas verificadas do Facebook, e as arestas, que representam "likes" mútuos entre duas páginas. Cada vértice é rotulado de acordo com sua categoria, que pode ser uma empresa (0), um governo (1), um político (2) ou um programa de televisão (3). Esses rótulos de categoria podem ser utilizados para realizar diversas análises envolvendo subgrafos do grafo.

Um caso para análise de subgrafos em um grafo de páginas do Facebook com rótulos de categoria é identificar todas as conexões entre políticos e companhias. Isso pode ser feito buscando por grafos do tipo clique, que são subgrafos onde todos os vértices estão conectados entre si.

Para encontrar grafos de clique com pelo menos dois vértices, podemos utilizar algoritmos de MPG para percorrer o grafo e identificar os subgrafos que atendem a esse critério. Além disso, podemos filtrar os resultados para incluir apenas subgrafos que contenham pelo menos um vértice com rótulo de "político" e pelo menos um vértice com rótulo de "companhia". Na Figura 22 é possível observar como podemos fazer essa análise em nossa ferramenta.

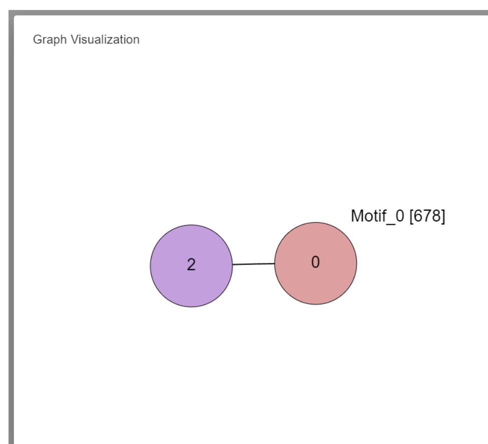
Figura 22 – Solução montada na Interface



Fonte: Produzido pelo Autor

É possível notar que nessa análise (Figura 23), foi possível contar 678 conexões de políticos com empresas via Facebook, essa informação pode ser um ponto de partida para uma pesquisa mais a fundo sobre essa relação entre companhias e políticos.

Figura 23 – Relação Políticos x Companhias



Fonte: Produzido pelo Autor

Existem diversas razões pelas quais alguém poderia querer contar os subgrafos que relacionam políticos e empresas em um grafo de páginas do Facebook com rótulos de categoria. Algumas dessas razões incluem:

- Identificação de possíveis conflitos de interesse: ao identificar todas as conexões entre políticos e empresas, é possível verificar se há políticos que têm relações estreitas com empresas que possam ter interesse em suas decisões. Essas conexões podem levantar suspeitas de corrupção e indicar possíveis conflitos de interesse que precisam ser investigados.

- Análise de redes políticas e econômicas: a contagem dos subgrafos que relacionam políticos e empresas pode ajudar a entender como as redes políticas e econômicas estão interligadas. Essa análise pode revelar padrões de poder e influência que podem ser úteis para entender como as decisões são tomadas e como a política e a economia estão interconectadas.

- Identificação de oportunidades de negócios: para empresas, a contagem dos subgrafos que relacionam políticos e empresas pode revelar oportunidades de negócios. Ao identificar políticos que estão conectados a empresas relevantes para seus negócios, as empresas podem buscar estabelecer parcerias ou alianças que possam ser benéficas para ambas as partes.

4.1.3 Mico

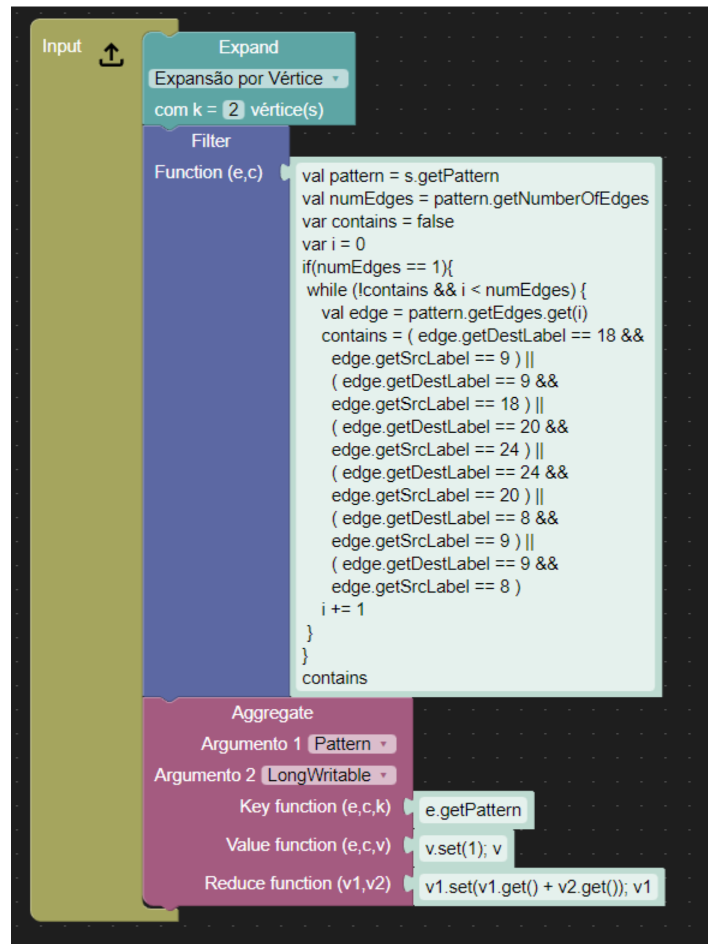
O grafo Mico é uma representação visual de dados da base de pesquisa da Microsoft Research. Neste grafo, os vértices representam autores e as arestas indicam coautoria em artigos científicos. Além disso, os rótulos nos vértices fornecem informações sobre as áreas de interesse de pesquisa desses autores.

Para essa análise vamos explorar a possibilidade de comparar 3 conexões entre 2 áreas diferentes no Grafo Mico da Microsoft Research. Essa análise pode nos ajudar a identificar qual par de áreas tem a maior "afinidade" em termos de coautoria em artigos científicos. Na Figura 24 vemos como podemos resolver esse problema utilizando a interface.

Ao comparar (Figura 25) essas conexões, podemos identificar qual par de áreas tem a maior "afinidade" em termos de coautoria em artigos científicos. Isso pode nos ajudar a entender quais áreas de pesquisa estão mais interligadas e quais áreas podem se beneficiar de uma maior colaboração entre seus pesquisadores. Nesse exemplo vemos que o par das áreas 8 e 9 apareceram muito mais que os outros dois selecionados.

Em resumo, a análise de conexões entre áreas diferentes no Grafo Mico da Microsoft Research pode nos fornecer informações valiosas sobre a interconexão de áreas de pesquisa e identificar áreas que podem se beneficiar de uma maior colaboração.

Figura 24 – Solução montada na Interface



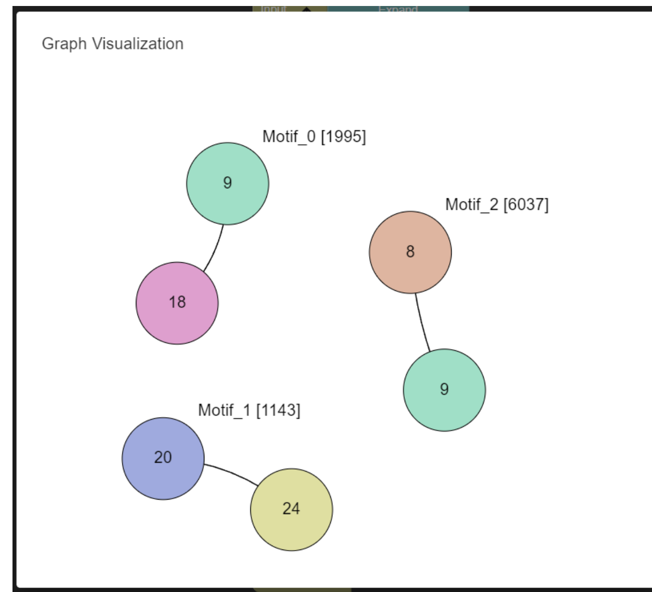
Fonte: Produzido pelo Autor

4.2 Análise do custo adicional

Nesta seção iremos analisar o custo que nossa interface teve sobre a forma original de se usar nossa ferramenta de MPG, no caso o Fractal, que seria executando uma aplicação já compilada via terminal. A avaliação experimental do custo adicional introduzido pela interface é uma área de grande importância em sistemas complexos de software. Em geral, essas interfaces e comunicações extras são necessárias para garantir que os módulos se comuniquem de forma eficiente e eficaz, mas também podem ter um impacto negativo no desempenho e na escalabilidade do sistema.

Para avaliar esse custo adicional, é necessário realizar testes em que as interfaces e comunicações extras são introduzidas em um sistema e os resultados são comparados com um sistema sem essas funcionalidades adicionais. Esses testes podem incluir a medição do tempo de execução, a utilização da CPU, o consumo de memória e outros fatores relevantes para o desempenho do sistema.

Figura 25 – Comparação entre 3 pares de áreas distintas



Fonte: Produzido pelo Autor

Para este trabalho vamos considerar que a máquina de teste será sempre a mesma, e em mesmas condições de uso. O critério que iremos avaliar aqui será o critério de tempo de execução, porém para trabalhos futuros pode ser interessante avaliar também custos de CPU e memória. Todos os nossos testes foram executados na seguinte configuração:

- **Sistema Operacional.** Ubuntu 20.04.6 LTS on Windows 10 x86_64;
- **Kernel.** 5.10.16.3-microsoft-standard-WSL2;
- **CPU.** AMD Ryzen 7 3700U with Radeon Vega Mobile Gfx (8 núcleos) @ 2.295GHz;
- **RAM.** 3GB para o subsistema WSL, 2GB para o SPARK;

4.2.1 Motifs com k vértices

Para nosso primeiro teste foi então montado a aplicação MPG que tem como tarefa encontrar padrões em subgrafos com k vértices. Para cada k foram feitas 10 execuções e depois retirada uma média simples agrupando pelo número de vértices, podemos ver abaixo na Tabela 2 o resultado que obtivemos, esse primeiro foi fora de nossa interface e executando um código já compilado via terminal.

Podemos observar que Motifs tem um comportamento exponencial, a medida que aumentamos o número de vértices em nossa expansão, temos um tempo de execução crescendo de forma exponencial.

Tabela 2 – Motifs, no terminal

Motifs, no terminal	
Nº Vértices	*Tempo Execução (s)*
2	1.30
3	1.16
4	1.53
5	5.38

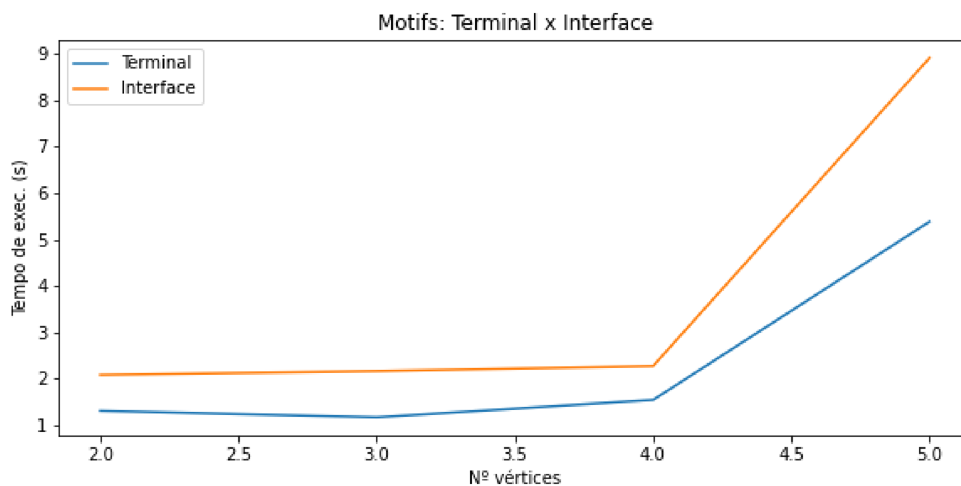
Também é possível comparar com o resultado da mesma aplicação, porém executando sobre a nossa plataforma construída. O resultado vemos na Tabela 3.

Tabela 3 – Motifs, na interface

Motifs, na interface	
Nº Vértices	*Tempo Execução (s)*
2	2.08
3	2.16
4	2.27
5	8.91

Podemos ver a comparação conforme a Figura 26 abaixo, fica claro que nossa interface adiciona um custo no tempo de execução do algoritmo o que era esperado. Entretanto, é importante destacar que esse aumento é mínimo e pode ser considerado insignificante quando comparado aos benefícios que a nova interface trouxe para a usabilidade da ferramenta.

Figura 26 – Comparação Motifs



Fonte: Produzido pelo Autor

4.2.2 Cliques com k vértices

Agora nosso objetivo é comparar como a aplicação Cliques se comporta em nossa interface comparativamente com um código já que foi compilado e rodando no terminal. Para cada k foram feitas execuções e depois retirada uma média simples agrupando pelo número de vértices, abaixo na Tabela 4 é possível observar como nossa aplicação se comportou quando executada no terminal. E comparativamente temos o resultado em nossa interface, conforme a Tabela 5.

Tabela 4 – Cliques, no terminal

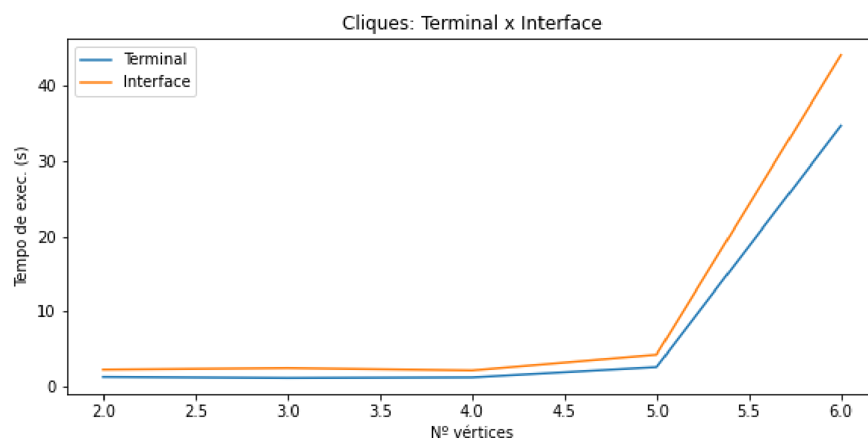
Cliques, no terminal	
Nº Vértices	*Tempo de exec (s)*
2	1.28
3	1.14
4	1.22
5	2.60
6	34.70

Tabela 5 – Cliques, na interface

Cliques, na interface	
Nº Vértices	*Tempo de exec (s)*
2	2.26
3	2.47
4	2.16
5	4.25
6	44.10

Com essas informações, é possível gerar a Figura 27 que mostra comparativamente como nossa interface desempenhou em relação nosso cenário base de executar a aplicação MPG, é possível observar novamente que nossa interface apesar adicionar tempo de execução, esse tempo não foi um aumento significativo.

Figura 27 – Comparação Cliques



Fonte: Produzido pelo Autor

5 Conclusão

Os resultados obtidos neste trabalho foram positivos, uma vez que o processamento de grandes quantidades de dados é um tema cada vez mais importante e presente em diversas áreas, e a democratização do acesso a tecnologias de análise de dados é um fator crítico para o sucesso dessas áreas. Dessa forma, a interface de programação visual proposta apresentou-se como uma solução viável e promissora para aprimorar a usabilidade e tornar mais acessível o desenvolvimento de aplicações de MPG para um público mais amplo.

Entendo que este trabalho contribui para a evolução das tecnologias de análise de dados, promovendo a democratização do acesso a essas ferramentas e ampliando o leque de usuários capazes de utilizá-las. A implementação da interface de programação visual mostrou que é possível tornar mais simples e intuitivo o desenvolvimento de aplicações de MPG, permitindo que mais pessoas possam explorar e utilizar as ferramentas de análise de dados de forma mais eficiente.

Além disso, cabe ressaltar que a implementação da interface de programação visual se concentrou na expansão por vértices, um dos principais problemas abordados pela mineração de padrões em grafos, e que a arquitetura proposta é capaz de viabilizar a construção de algoritmos de MPG de diversos tipos. Nesse sentido, um próximo passo seria a implementação de outras formas de expansão, como por arestas ou padrões específicos, de forma a ampliar ainda mais a versatilidade da ferramenta.

Ademais, é importante destacar que a arquitetura proposta precisa de refinamentos adicionais para que funcione de forma mais escalável, sobretudo quando acessada por múltiplos usuários simultaneamente. Para atingir esse objetivo, é necessário otimizar a infraestrutura da aplicação e melhorar a forma como os recursos computacionais são gerenciados.

Por fim, é importante destacar que a interface de programação visual desenvolvida neste trabalho tem potencial para ser aplicada em diversas áreas que requerem o processamento de grandes quantidades de dados, inclusive em disciplinas como "Análise de Midías Sociais" fornecendo uma interface que pode auxiliar os alunos na compreensão das análises possíveis de serem feitas dado um grafo de entrada. Portanto, um próximo passo seria a publicação da ferramenta em um link na web, a fim de disponibilizar a interface para um público ainda mais amplo e fomentar novas possibilidades de uso.

Referências

DIAS, V. et al. Fractal: A general-purpose graph pattern mining system. In: *Proceedings of the 2019 International Conference on Management of Data*. New York, NY, USA: Association for Computing Machinery, 2019. (SIGMOD '19), p. 1357–1374. ISBN 9781450356435. Disponível em: <https://doi.org/10.1145/3299869.3319875>. Citado na página 22.

EASLEY, D.; KLEINBERG, J. *Networks, Crowds, and Markets: Reasoning about a Highly Connected World*. Cambridge University Press, 2010. ISBN 9781139490306. Disponível em: <https://books.google.com.br/books?id=atfCl2agdi8C>. Citado na página 14.

ELSEIDY, M. et al. Grami: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow.*, VLDB Endowment, v. 7, n. 7, p. 517–528, mar. 2014. ISSN 2150-8097. Disponível em: <http://dx.doi.org/10.14778/2732286.2732289>. Citado na página 47.

HOFFMAN, F.; KRASLE, D. *Fraud detection using network analysis*. 2015. Patent No. EP2884418A1, Filed September 1st., 2014, Issued June 17th., 2015. Citado na página 14.

JOHNSSON, B. A.; MAGNUSSON, B. Towards end-user development of graphical user interfaces for internet of things. *Future Generation Computer Systems*, v. 107, p. 670–680, 2020. ISSN 0167-739X. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167739X17321660>. Citado 2 vezes nas páginas 20 e 21.

JOST, B. et al. Graphical programming environments for educational robots: Open roberta - yet another one? In: *2014 IEEE International Symposium on Multimedia*. [S.l.: s.n.], 2014. p. 381–386. Citado 2 vezes nas páginas 12 e 19.

KUHAIL, M. A. et al. Characterizing visual programming approaches for end-user developers: A systematic review. *IEEE Access*, IEEE, v. 9, p. 14181–14202, 2021. ISSN 2169-3536. Citado 2 vezes nas páginas 19 e 21.

MILO, R. et al. Network motifs: Simple building blocks of complex networks. *Science*, v. 298, n. 5594, p. 824–827, 2002. Disponível em: <https://www.science.org/doi/abs/10.1126/science.298.5594.824>. Citado 2 vezes nas páginas 14 e 22.

PASTERNAK, E.; FENICHEL, R.; MARSHALL, A. N. Tips for creating a block language with blockly. In: . Raleigh, NC, USA: IEEE, 2017. p. 21–24. ISBN 978-1-5386-2481-4. Citado na página 29.

RAO, A.; BIHANI, A.; NAIR, M. Milo: A visual programming environment for data science education. In: *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. [S.l.: s.n.], 2018. p. 211–215. Citado 2 vezes nas páginas 19 e 20.

READ, K. E. Cultures of the central highlands, new guinea. *Southwestern Journal of Anthropology*, JSTOR, p. 1–43, 1954. Citado na página 13.

RODIANI, R. *Visual Programming Interface for Graph Pattern Mining*. 2023. Disponível em: <https://github.com/ricardoRodiani/vpl-fractal>. Citado na página 47.

ROZEMBERCZKI, B.; ALLEN, C.; SARKAR, R. *Multi-scale Attributed Node Embedding*. 2019. Citado 2 vezes nas páginas 47 e 50.

THAMSEN, L. et al. Visually programming dataflows for distributed data analytics. In: *2016 IEEE International Conference on Big Data (Big Data)*. [S.l.: s.n.], 2016. p. 2276–2285. Citado na página 21.

UGANDER, J.; BACKSTROM, L.; KLEINBERG, J. M. Subgraph frequencies: mapping the empirical and extremal geography of large graph collections. In: SCHWABE, D. et al. (Ed.). *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013*. [S.l.]: International World Wide Web Conferences Steering Committee / ACM, 2013. p. 1307–1318. Citado na página 14.

Apêndices

Figura 28 – Passo 1 Cliques 3

Figura 29 – Passo 2 Cliques 3

Figura 30 – Passo 3 Cliques 3

Figura 31 – Passo 1 Motifs 3

Figura 32 – Passo 2 Motifs 3

Figura 33 – Passo 3 Motifs 3