UM FRAMEWORK PARA IMPLEMENTAÇÃO DE ALGORITMOS DE PARTICIONAMENTO DE GRAFOS

Roberto Ribeiro Rocha
Universidade do Vale do Sapucaí
Av. Pref. Tuany Toledo, 470 – Fátima – CEP: 37.550-000 – Pouso Alegre-MG
rrocha.roberto@gmail.com

Edmilson Marmo Moreira
Universidade Federal de Itajubá
Av. BPS, 1303 – Pinheirinho – CEP: 37.500-903 – Itajubá-MG
edmarmo@unifei.edu.br

Otávio Augusto Salgado Carpinteiro
Universidade Federal de Itajubá
Av. BPS, 1303 – Pinheirinho – CEP: 37.500-903 – Itajubá-MG
otavio@unifei.edu.br

RESUMO

Este artigo apresenta um *framework* para auxiliar o desenvolvimento de programas que utilizem algoritmos para o particionamento de grafos, permitindo a manipulação das informações tanto em memória quanto em um banco de dados orientado a grafos. O *framework* foi projetado para trabalhar com algoritmos *2-way*, *k-way* e multiníveis, dando suporte para que os algoritmos utilizem as estruturas do banco Neo4J. Esta característica permite que as implementações dos algoritmos sejam realizadas independentemente do recurso utilizado (memória ou disco), fornecendo ao pesquisador uma estrutura de dados genérica e flexível.

PALAVRAS CHAVE. Grafos, Particionamento de grafos, *Framework*, Bancos de dados orientados a grafos.

Área princial: TAG – Teoria e Algoritmos em Grafos

ABSTRACT

This paper presents a framework for helping the development of programs which use algorithms for graph partitioning, enabling information handling both in memory and in a graph database. The framework was designed to work with 2-way, k-way and multilevel algorithms, providing support to the algorithms to make use of the Neo4J database structures. This feature allows the implementations of the algorithms are performed independently of the resource used (memory or disc), providing the researcher a generic and flexible data structure.

KEYWORDS. Graphs, Graph partitioning, Framework, Graph databases.

Main area: TAG - Theory and Algorithms in Graphs

1. Introdução

A identificação de conjuntos de indivíduos e seus relacionamentos forçou a busca por novas formas de classificação e agrupamento de informações, facilitando, assim, encontrar soluções para problemas em diversas áreas, tais como: biologia, física, química, redes sociais, telecomunicações, análise de imagens, engenharias, balanceamento de carga em computação paralela e/ou distribuída, etc. Nestas áreas do conhecimento, a detecção de estruturas de comunidades é importante, pois revelam certos fenômenos, muitas vezes ocultos.

Por sua vez, muitos desses problemas da vida real podem ser representados através de grafos, permitindo identificar soluções através da aplicação de técnicas de particionamento. Assim, pode-se utilizar várias abordagens para a análise da inter-relação entre os objetos de um determinado conjunto de dados, tais como, minimização do corte (KERNIGHAN; LIN, 1970), conectividade, computação de centróides (DUTOT; OLIVIER; SAVIN, 2011), cortes naturais (DELLING et al., 2011) e medidas de modularidade (NEWMAN; GIRVAN, 2003).

Um ponto relevante na utilização de grafos na solução desses problemas de classificação, entre outros, é a quantidade de elementos que serão tratados. As estruturas de dados usualmente utilizadas para representar um grafo em memória possuem, claramente, limitações. Neste sentido, o uso de mecanismos de armazenamento persistentes é essencial para o tratamento de grandes quantidades de dados.

Neste contexto, este artigo apresenta um *framework* cujo objetivo é auxiliar usuários que utilizam grafos para representar as informações de relacionamento em suas aplicações e necessitam realizar particionamento nos seus conjuntos de dados. Considerando o grande crescimento da quantidade de informações que as aplicações atuais manipulam, este *framework* possibilita o tratamento dessas informações em memória ou, principalmente, em disco.

Este artigo esta estruturado da seguinte forma: a seção 2 apresenta uma breve introdução sobre particionamento de grafos e a seção 3 apresenta as principais características de um banco de dados orientado a grafos. A seção 4 revisa alguns algoritmos clássicos de particionamento, que estão disponíveis no *framework*, e a seção 5 discute a modelagem do *framework*. A seção 6 apresenta uma discussão sobre o seu uso e, finalmente, a seção 6 conclui o artigo.

2. Particionamento de Grafos

Os grafos têm um papel importante em várias áreas da ciência devido ao fato de permitirem que problemas do mundo real sejam generalizados em estruturas bem definidas, facilitando o processo de particionamento e obtendo-se assim resultados importantes na classificação das informações que eles representam.

Um *grafo G = (V,E)* consiste de um conjunto finito *V* de vértices e um conjunto finito *E* de arestas sendo que cada elemento *E* possui um par de vértices que estão conectados entre si e possui um peso *P*.

Existem várias formas de representar um grafo computacionalmente, dentre as mais utilizadas cita-se: matrizes de incidência e adjacência, listas de adjacências e objetos. Na representação utilizando objetos, são utilizados os recursos das linguagens orientadas a objetos para organizar os vértices e arestas do grafo, utilizando atributos de classes para armazenar as informações dos relacionamentos entre os objetos envolvidos. A figura 1 ilustra este tipo de representação. Um objeto vértice possui uma lista de objetos da classe aresta. Um objeto aresta possui dois atributos da classe vértice, um indicando a ponta *a* e outro indicando a ponta *b*, facilitando a navegação pela aresta. Essa aresta pode possuir atributos, por exemplo, seu peso.

O problema do particionamento de um grafo consiste em dividir esse grafo em *i* subconjuntos de vértices de maneira que o corte de arestas seja minimizado e que cada subconjunto possua uma quantidade equilibrada de vértices. O corte de arestas, segundo Kernighan e Lin (1970), é a soma dos pesos das arestas nas quais seus vértices estejam em diferentes conjuntos. Para grafos cujas arestas não possuam peso, ele é considerado unitário.

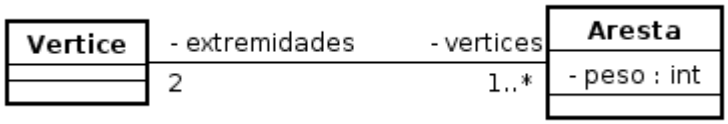

Figura 1. Classes representando vértices e arestas

Um particionamento pode ser classificado quanto à quantidade de partições (bipartição e particionamento *k-way*) e também quanto à heurística de busca de soluções, mostradas a seguir:

- **locais**: fazem a busca da solução utilizando os vértices vizinhos aos vértices que estão sendo processados em um dado momento. Estes possuem uma gama maior de variações e técnicas para o particionamento:
  - **troca de vértices**: melhoram um particionamento fazendo a troca de vértices entre as partições. Entre esses métodos, os mais conhecidos são os métodos de Kernighan e Lin (1970), Fiduccia e Mattheyses (1982) e suas variações;
  - **gulosos**: criam conjuntos de vértices que são iniciados a partir de vértices sementes e utilizam técnicas gulosas para avançar no grafo. Como exemplo, cita-se o método *Greedy-Kway* (JAIN; SWAMY; BALAJI, 2007);
  - **divisivos**: removem arestas chave, até deixar o grafo desconexo, obtendo assim as partições desejadas, que é o caso do método proposto por Newman e Girvan (2003).
  - **aglomerativos**: agrupam vértices considerados próximos, incorporando-os ao conjunto apropriado, conforme os métodos de Blondel et al. (2008) e Delling et al. (2009);
  - **difusivos**: utilizam técnicas de difusão de líquido ou gases, de forma que, a partir de vértices sementes, diferentes líquidos são injetados em cada vértice que são transferidos para outros vértices até se estabilizarem (PELLEGRINI, 2007) e (GEHWEILER; MEYERHENKE, 2010);
- **globais**: utilizam do conhecimento global de um grafo através da teoria espectral dos grafos, iniciada pelo método de Fieldler (FIEDLER, 1973) ou métodos iterativos (HERNÁNDEZ et al., 2007). Algumas variações foram desenvolvidas, como por exemplo, o método *K-Cut* (RUAN; ZHANG, 2007) e algumas delas foram avaliadas por Nascimento (2010);
- **multiníveis**: iniciados por Karypis e Kumar (1995), contraem o grafo através do emparelhamento de arestas, particionam e projetam o particionamento de volta ao grafo original;
- **métodos mistos**: utilizam da combinação dos métodos anteriores aproveitando as melhores características de cada um para definir um novo método, como por exemplo o método apresentado por Bonatto e Amaral (2010).

3. Banco de Dados Orientados a Grafos

Em várias classes de aplicações surgem dificuldades no uso de bancos de dados relacionais. Assim, foram criados outros tipos de bancos de dados chamados de *NoSQL*, um acrônimo para *Not only SQL*. Um modelo implementado é o banco de dados orientado a grafos. Neste tipo de banco, há o armazenamento dos vértices e arestas sem o uso de tabelas, permitindo a execução de consultas rápidas através de travessias no grafo, acessando somente os vértices pertencentes ao escopo da consulta e evitando *joins* caros, muito utilizados nos bancos relacionais.

O *framework*, aqui apresentado, utiliza como banco de dados orientado a grafos o Neo4J (NEO4J, 2015), que possui a licença *GPLv3*. O Neo4J possui vários elementos nos quais são implementados toda a estrutura de dados de vértices e arestas, algoritmos e gerenciamento de índices. Os relacionamentos (arestas) organizam os vértices e ambos possuem seus atributos que são informações do mundo real ou informações de controle interno para qualquer algoritmo que deseja trabalhar com esses elementos. As travessias navegam no grafo para identificar caminhos, executando assim algum algoritmo. Um exemplo de utilização do Neo4J é o sistema de roteirização apresentado por Domingos et al. (2012).

As classes e interfaces mais importantes do Neo4J utilizadas no *framework* são:

- **GraphDatabaseService**: Interface que provê o acesso para uma instância do Neo4J, fornecendo serviços de criação e recuperação de vértices e arestas entre outros.
- **EmbeddedGraphDatabase**: Implementação de *GraphDatabaseService* para uso embutido no programa Java, para a criação e uso do banco em um diretório local da execução.
- **Index**: Interface Java para criação e uso de índices baseados em pares chave e valor, que podem ser criados tanto para vértices quanto para arestas.
- **Transaction**: Interface que permite o manuseio de transações por meio de programação.
- **Node**: Representa o vértice.
- **Relationship**: Representa a aresta ligando dois vértices.

4. Algoritmos de Particionamento

Os algoritmos de particionamento de grafos foram iniciados por Kernighan e Lin (1970), para colocar componentes eletrônicos em placas de circuitos impressos, minimizando o número de conexões entre as placas.

Porém, devido ao fato do problema de particionamento não possuir uma solução trivial, sendo um problema combinacional, as soluções propostas são heurísticas que tentam aproximar a solução final da melhor solução. A seguir serão mencionados quatro algoritmos de particionamento que suportaram a criação do *framework*.

4.1. Heurística KL - Kerningan-Lin

O objetivo do algoritmo, proposto por Kernighan e Lin (1970), é criar um particionamento inicial com dois conjuntos arbitrários *A* e *B* a partir de um grafo *G*, e tentar diminuir o custo externo inicial *T* por uma série de trocas entre os vértices dos subconjuntos de *A* e *B*.

Essa heurística identifica um elemento de cada lado da partição (*2-way*) de forma que ao fazer a troca desses elementos, o custo do corte seja reduzido. Assim, a principal questão é fazer a escolha adequada desses elementos, que é baseada no custo externo e custo interno de um vértice, que corresponde à soma dos pesos das arestas deste vértice para a outra partição e para sua própria partição, respectivamente. Após a identificação dos custos, é calculada, para cada vértice, a diferença *D* entre os custos externo e interno de cada vértice.

Para decidir o par de vértices que deve ser trocado entre as partições, calcula-se o valor do ganho de cada par, ou seja, a redução do custo, para os pares de vértices, expresso por:

$$g(a,b) = D(va) + D(vb) - 2c(va,vb)$$

onde *c* é o peso da aresta entre os vértices *va* e *vb*. Valores negativos de *g* podem indicar que a troca dos vértices faça a solução escapar de um mínimo local, melhorando o resultado final do particionamento. Em seguida, identifica-se o par que produz o maior ganho e armazena-o temporariamente. Então, o algoritmo recalcula os valores de *D* para os elementos, vizinhos ao par escolhido, que ainda não foram processados na iteração atual.

O algoritmo repete a escolha do par e os cálculos de D e do ganho dos vértices restantes, até todos os vértices terem sido analisados. Com todos os pares obtidos, escolhe-se k pares para maximizar a soma parcial S dos ganhos calculados. Se $S > 0$, então uma redução no custo com o valor de S pode ser obtida trocando-se os k pares.

4.2. Algoritmo FM - Fiduccia e Mattheyses

Esta foi a segunda heurística desta área de grafos, proposta por Fiduccia e Mattheyses (1982), que visa melhorar de modo iterativo um particionamento. Diferentemente do KL, o algoritmo move um vértice por vez, de uma partição para outra, para minimizar o tamanho do corte final entre as partições, mantendo um balanceamento do peso dos vértices por partição. O vértice a ser movido, chamado de célula base, é escolhido baseado em um critério de balanceamento e no efeito produzido por sua mudança no tamanho do corte atual. Este efeito corresponde ao número de arestas pelo qual o corte diminuiria, devido à mudança de partição desse vértice. O critério de balanceamento evita que todos os vértices migrem de um conjunto para outro. Mesmo se o ganho não for positivo, o vértice é movido, com a expectativa de que o movimento irá permitir que o algoritmo saia de um mínimo local.

Os autores sugerem que os vértices sejam colocados em uma fila de prioridade, chamada de *bucket*, uma para cada partição, facilitando a remoção, busca e inserção de vértices livres a cada mudança de seu ganho. Aproveitando a ideia de Kernighan e Lin (1970), após os movimentos, o melhor particionamento encontrado durante o passo é obtido como saída do passo.

Os vértices já movidos são marcados, para evitarem de ficar migrando de uma partição para outra indefinidamente, sendo que somente vértices livres podem fazer um movimento em cada passo do algoritmo.

4.3. Bipartição Multinível

A principal ideia do algoritmo proposto por Karypis e Kumar (1995) é diminuir o tamanho do grafo original, obtendo um grafo equivalente reduzido e minimizando o esforço de particionamento, que é executado em três fases bem definidas:

1. **Contração**: o grafo original é transformado em uma sequência de grafos menores, cada um com menos vértices, preservando as propriedades do grafo original.
2. **Particionamento**: consiste na produção de uma bissecção de alta qualidade (isto é, pequeno corte de arestas), pois, neste momento, o grafo possui uma quantidade reduzida de vértices.
3. **Expansão**: processo que consiste em projetar a partição mais contraída de volta para o grafo original, indo através das partições intermediárias. Cada particionamento intermediário pode não necessariamente ser local, pois possui um grau maior de liberdade. Assim, pode ser utilizada uma heurística de refinamento local para melhorar ainda mais o particionamento.

Os autores também sugerem um algoritmo de refinamento local, baseado em Kernighan e Lin (1970), chamado refinamento BKL (*Boundary Kernighan-Lin*), onde é feita a troca de pares de vértices que participam do corte atual, após a expansão.

4.4. Greedy K-Way

Esse algoritmo *k-way*, definido por Jain, Swamy e Balaji (2007), faz o particionamento de um grafo iniciando com a escolha aleatória de k vértices sementes e, no decorrer da execução do algoritmo, os vértices remanescentes são adicionados alternadamente a cada partição. Assim, a cada estágio, o vértice adicionado será aquele que resulta em um aumento mínimo no corte. Esse processo se baseia em abordagens gulosas de particionamento *2-way* que definem regras

para escolha do vértice e partição mais apropriados.

O método se baseia na definição de vizinhança ou fronteira, que corresponde ao conjunto de vértices candidatos para serem incluídos no próximo passo, nos quais utilizam as arestas que conectam esses vértices aos vértices que já pertençam a alguma partição. O vértice *v* escolhido para ser adicionado à partição *p* é aquele no qual a inclusão dele na partição maximize a quantidade de arestas internas e minimize a quantidade de arestas externas (arestas do corte), procedendo gulosamente até consumir todos os vértices.

5. A modelagem do *framework*

O *framework* apresentado neste trabalho auxilia o desenvolvimento de programas de particionamento de grafos, oferecendo uma infraestrutura de representação do grafo, tanto em memória quanto em um banco de dados orientado a grafos. Para facilitar a implementação dos algoritmos de particionamento, é necessário compor os principais componentes em uma arquitetura de *software*, que é uma estruturação através de componentes ou módulos. Neste contexto, um ambiente de particionamento de grafos pode ser articulado em uma arquitetura conforme ilustra a figura 2.

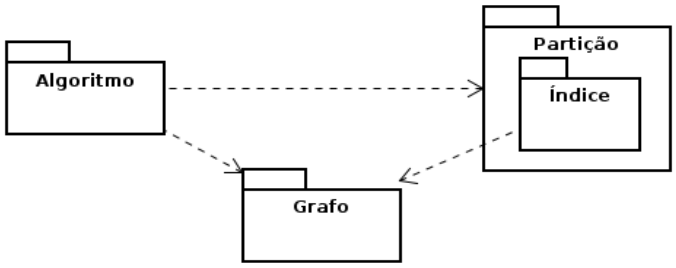

Figura 2. Arquitetura de componentes de um ambiente de particionamento de grafos

O componente “Algoritmo” é responsável pela lógica de execução do algoritmo desejado, bem como manter as estruturas de dados pertencentes à sua execução interna. O componente “Partição” é responsável por manter as informações sobre o particionamento em si, tais como, quais vértices pertencem a certa partição, quais arestas pertencem ao corte atual e o valor do corte atual. Esse componente provê suporte para o componente “Algoritmo” de forma que este se preocupe somente com a lógica necessária para definir em qual partição o vértice deve ser adicionado ou removido.

O componente “Grafo” possui as classes e interfaces genéricas que representam o grafo em si, com vértices arestas e seus atributos, além dos serviços oferecidos pelo grafo e outras classes auxiliares. Este componente oferece classes especializadas que implementam os serviços definidos para duas formas de acesso às informações do grafo: acesso das informação do grafo no banco de dados orientado a grafos Neo4j e acesso das informações do grafo utilizando a memória principal.

Já no componente “Partição”, se encontram as estruturas e operações de atualização e cálculo do valor do corte, além de classes de indexação de vértices, contidas no subcomponente “Índice” que facilita a obtenção do conjunto atual de vértices de uma dada partição, além da inserção e da remoção de vértices de seus respectivos índices.

Com este modelo de componentes, o *framework* alcança um bom nível de flexibilidade e abstração, permitindo combinar diversos algoritmos utilizando uma estrutura genérica de acesso às informações do grafo de forma simples e eficiente. É possível, por exemplo, fazer uma implementação do algoritmo KL (KERNIGHAN; LIN, 1970), utilizando um grafo contido em um arquivo texto, ou uma implementação do algoritmo *Greedy K-way* (JAIN; SWAMY; BALAJI, 2007) utilizando um grafo que esteja armazenado no banco de dados Neo4j.

A seguir será apresentada a modelagem do *framework* em diagramas de classes da UML (*Unified Modeling Language*).

5.1. O pacote Grafo

Através da descrição das classes do modelo desenvolvido, os principais elementos específicos e genéricos do *framework*, e seus relacionamentos, serão apresentados e discutidos, com a intenção de facilitar o entendimento dos recursos disponíveis para o desenvolvimento de outros algoritmos de particionamento.

O principal objetivo é proporcionar soluções para usuários que desejam implementar algoritmos de particionamento já existentes ou realizar experimentos de novas ideias. O *framework* oferece recursos para manipulação do grafo tanto em memória quanto no banco Neo4J, garantindo a integridade das informações manipuladas assim como o reaproveitamento do mesmo grafo na execução de vários algoritmos. Além dessas características, o *framework* ainda pode ser estendido para trabalhar com outros bancos de dados orientados a grafos.

Como pode ser visto na figura 3, os principais elementos deste componente são as interfaces NodeWrapper e EdgeWrapper e a classe abstrata GraphWrapper. Essa abstração proporciona uma maior flexibilidade para o usuário do *framework*. É a partir desta estrutura que o algoritmo de particionamento deve fazer o acesso aos elementos do grafo.

A classe GraphWrapper é um contêiner que provê métodos para manter os objetos internos do grafo como um todo, fornecendo vários serviços relativos à criação e remoção de vértices e arestas através dos métodos createNode() e createEdge() e obtenção da lista de todos os vértices e todas as arestas, feita pelos métodos getAllNodes() e getAllEdges(). Também é possível obter um vértice através de seu identificador, utilizando o método getNode(long id). Estes métodos são abstratos para permitir que os detalhes de armazenamento, leitura e manipulação sejam feitos por suas subclasses, GraphDB e GraphMem. Outras operações concretas também foram definidas para saber a quantidade de vértices existentes no grafo, utilizando o método getSizeNodes() e para fazer o carregamento do grafo desejado, através do método readGraph(). Cada implementação específica implementa suas próprias formas de armazenar os valores de propriedades correspondentes ao objeto desejado.

A classe GraphDB possui a implementação dos métodos definidos por GraphWrapper de forma que toda a operação feita no grafo seja efetivada no banco de dados Neo4J, acessado através do parâmetro graphFileName especificado no construtor, através de uma instância da classe GraphDatabaseService, que possui métodos para criar e remover vértices e arestas do banco, recuperar um *iterator* para todos os vértices ou todas as arestas existentes, manipular índices criados a partir de vértices ou arestas e manipular as transações do banco de dados.

A classe GraphMem possui a implementação de seus métodos para manter em memória os objetos que compõem a estrutura do grafo. A leitura do grafo é feita a partir do arquivo especificado no parâmetro graphFileName do construtor.

Para permitir o uso de transações, foi definida a interface TransactionInterface, na qual define os serviços de início e fim de uma transação, bem como a ocorrência de falha ou o sucesso da mesma. Estes métodos são implementados pelas classes GraphDB e GraphMem. A classe GraphDB delega as chamadas a uma instância de org.neo4j.graphdb.Transaction e, por sua vez, a classe GraphMem não trata situações transacionais, pois qualquer execução implica no carregamento do grafo original novamente.

Devido ao volume de informações esperado para ser processada pelos algoritmos, foi definida também a classe TransactionControl que tem por objetivo iniciar a transação, através do método beginTransaction(), fazer uma operação para efetivar as alterações intermediárias caso a quantidade de operações atinjam um valor pré-definido, através do método intermediateCommit(), e finalizar a transação, com sucesso ou falha, usando os métodos commit() e rollback() respectivamente. Objetos dessa classe devem ser utilizados onde existir uma estrutura de repetição modificando algum atributo interno de um vértice ou aresta, pois casos onde há muitas modificações em uma mesma transação podem extrapolar a quantidade de memória disponível.

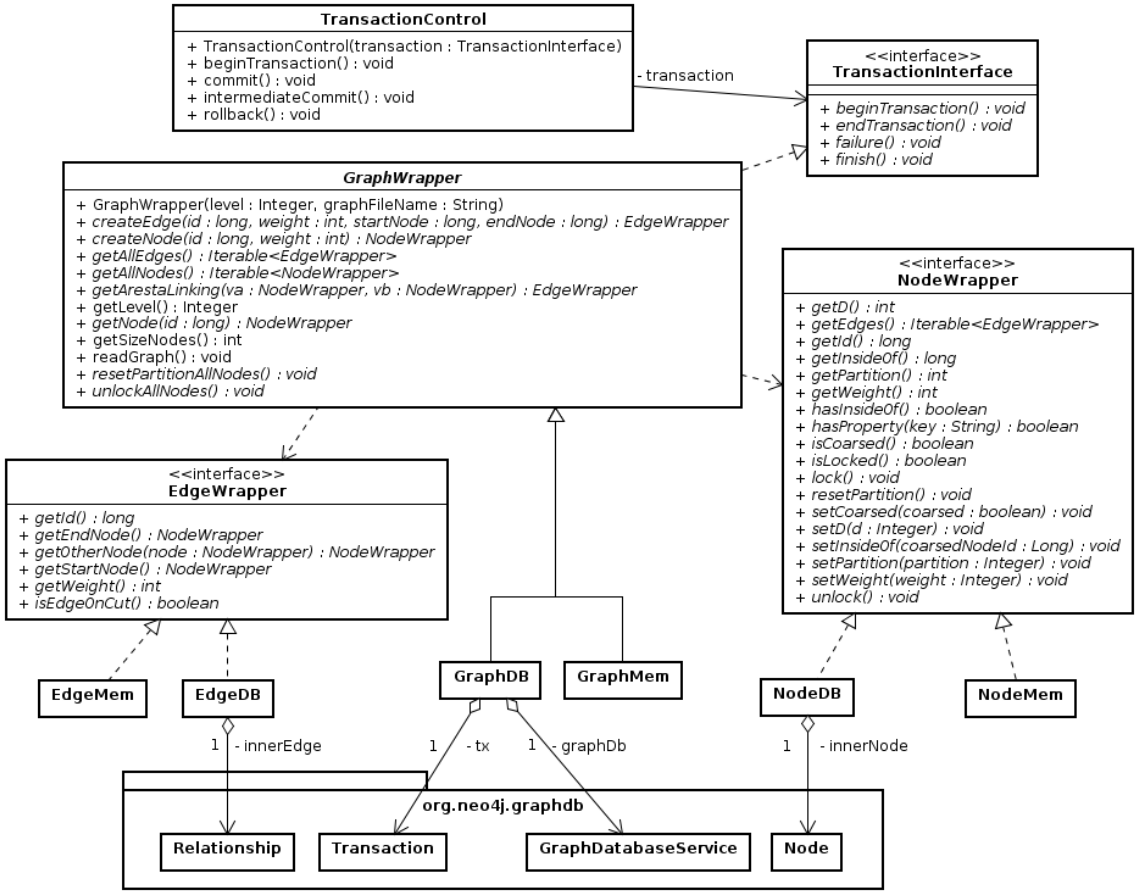

Figura 3. Abstração da estrutura do grafo - memória X banco de dados

A interface EdgeWrapper define os métodos pertinentes ao comportamento de uma aresta, dentre os quais destacam-se getWeight() que retorna seu peso, getOtherNode(NodeWrapper node) que retorna o vértice correspondente à ponta oposta ao vértice solicitado, isEdgeOnCut() que indica se a aresta está no corte ou não. E, finalmente, getStartNode() e getEndNode() que retornam o vértice de cada uma das extremidades da aresta. Como classes concretas, foram definidas duas subclasses, EdgeDB e EdgeMem, para implementar as operações de uma aresta no banco de dados e em memória, respectivamente.

A classe EdgeDB possui o atributo innerEdge, do tipo org.neo4j.graphdb.Relationship, para o qual delega todas os métodos definidos em EdgeWrapper, tratando apropriadamente a implementação do método isEdgeOnCut() que utiliza o número da partição dos vértices da aresta em questão. A classe EdgeMem mantém as informações da aresta através de seus atributos, para fornecer as informações definidas por EdgeWrapper.

Similarmente à classe EdgeDB, a classe NodeDB mantém uma instância de innerNode do tipo org.neo4j.graphdb.Node, para o qual delega todas os métodos definidos em NodeWrapper. Já a classe NodeMem mantém as informações do vértice em seus atributos.

A partir da classe GraphWrapper e das interfaces NodeWrapper e EdgeWrapper, podem ser criadas subclasses concretas que implementem outras formas de armazenamento de um grafo, deixando o código do algoritmo independente da forma de manipulação do grafo e permitindo ao usuário do *framework* adaptar as estruturas desses elementos para os dados específicos que ele deseja manipular durante a execução de seu algoritmo. Finalmente, o mesmo algoritmo pode utilizar quaisquer das duas opções fornecidas por este *framework*, bem como utilizar outras soluções derivadas deste.

5.2. O pacote Particionamento

Outro recurso oferecido é a facilidade de se trabalhar com as informações de particionamento em si, utilizando classes que auxiliam a manutenção dessas estruturas. As classes e interfaces deste pacote permitem que as tarefas relativas ao particionamento sejam facilitadas ao usuário do *framework*. A Figura 4 apresenta o diagrama de classes da solução proposta.

Neste componente, existem duas classes abstratas que são a base para o entendimento do diagrama: Partition e AbstractPartitionIndex. Primeiramente, a classe Partition é responsável por manter o atributo *index*, do tipo AbstractPartitionIndex, com as informações internas sobre um dado particionamento. Seu construtor permite que seja definido o número *k* de partições, aumentando a flexibilidade do *framework*, podendo, por exemplo, ser utilizado pelo algoritmo *Greedy K-way*. O parâmetro level, torna possível o uso desta classe por métodos multiníveis. O método createBestPartition() instancia e preenche as informações da partição atual em um novo objeto da subclasse BestPartition, designado para armazenar a melhor partição alcançada até um certo momento da execução do algoritmo. O restante de seus métodos delega as chamadas para o objeto *index*, que serão detalhados a seguir. A classe BestPartition também permite, ao usuário do *framework*, salvar as informações do particionamento em um arquivo texto, através do método exportPartitions(fileName).

A classe AbstractPartitionIndex provê os serviços necessários para manutenção do particionamento, isto é, inclusão e remoção de vértices de uma dada partição, aqui chamada de *set* e a consulta dos vértices e da quantidade de vértices de uma determinada partição. Essas funções são definidas por métodos abstratos, pois dependem se o grafo está em memória ou no banco de dados. Os métodos concretos para inserir, remover ou atualizar um vértice em uma partição, fazem cada um, uma chamada ao seu respectivo método abstrato e logo após atualizam o estado do corte de arestas, através de chamadas à sua superclasse EdgeCache. Esta classe foi criada com o intuito de manter a lista de arestas que estão no corte, e para isso possui os métodos para incluir e excluir arestas, recuperar a lista atual de arestas do corte e retornar o valor do peso das arestas do corte.

A classe PartitionIndexDB implementa os serviços de sua interface, fazendo a indexação dos vértices utilizando o próprio recurso de indexação interna do banco de dados Neo4J, representado por um objeto que implementa a interface org.neo4j.graphdb.index.Index. Já a classe PartitionIndexMem implementa os métodos mantendo uma lista interna de objetos da classe NodeCache, onde cada objeto desta lista representa uma partição, no qual cada partição possui uma lista de vértices, representado pelo atributo nodes.

6. Uso do *framework*

Esta seção discute a utilização do *framework* por parte do usuário para o desenvolvimento de algoritmos de particionamento.

Para facilitar o entendimento, a figura 5 apresenta um diagrama de classes com a implementação do algoritmo de Kernighan e Lin (1970), onde o código ainda não especifica qual será a estrutura interna de armazenamento do grafo.

Neste diagrama, a classe KL utiliza as interfaces NodeWrapper e EdgeWrapper, bem como as classes abstratas GraphWrapper e AbstractPartitionIndex para poder fazer os cálculos dos custos e de ganho, além de escolher e efetuar a troca dos pares de vértices entre as partições. As operações sobre o particionamento é feito através da classe TwoWayPartition, que inicializa os atributos de sua superclasse Partition para trabalhar com duas partições. As operações especializadas dessa subclasse consistem em criar uma partição inicial tanto fixa quanto aleatória e efetuar a troca dos pares de vértices.

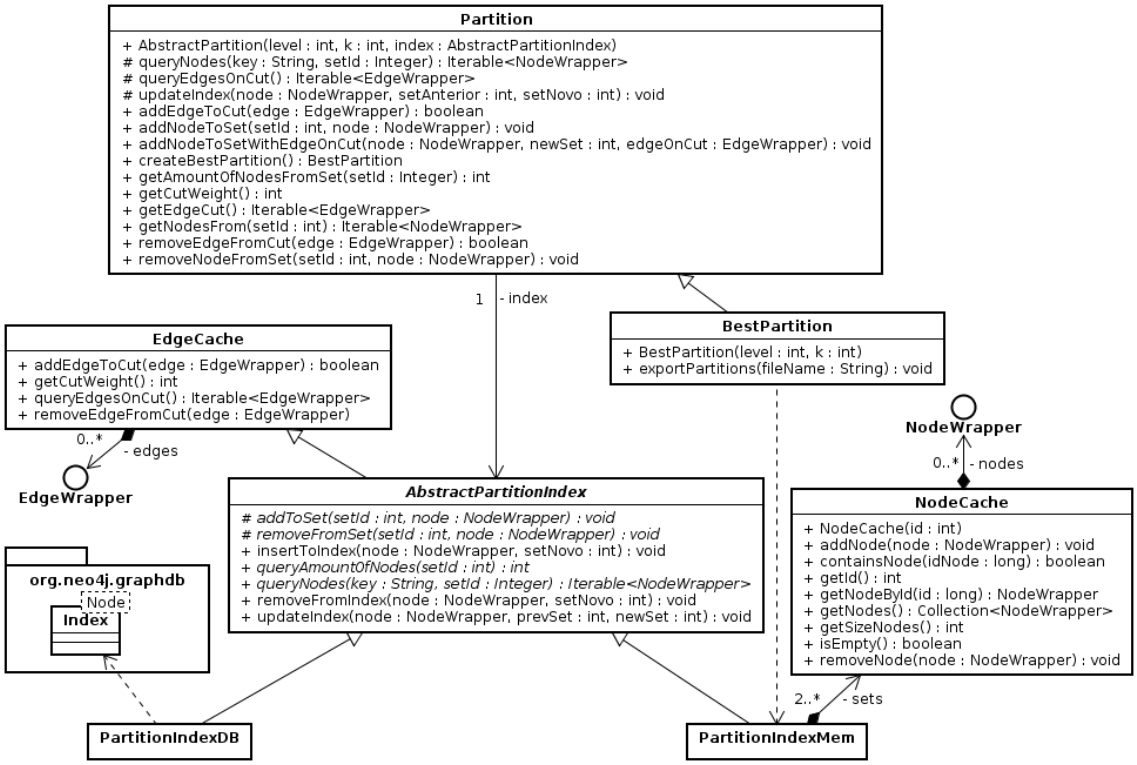

Figura 4. Classes de particionamento

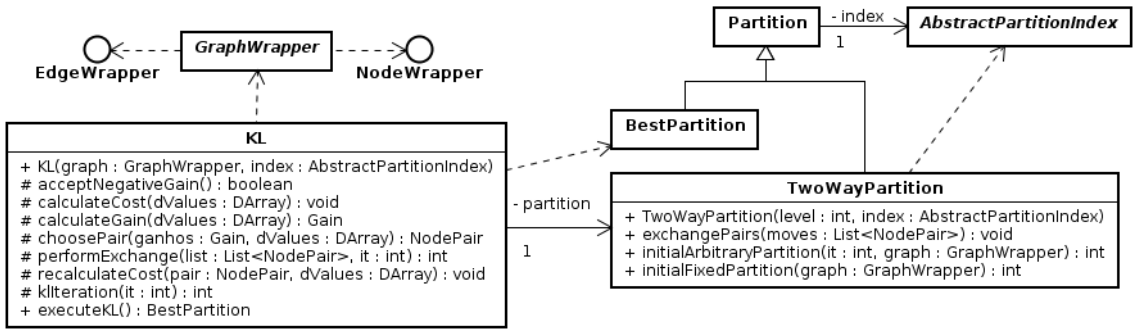

Figura 5. Implementação do algoritmo KL

A classe KL possui seu construtor com dois parâmetros, graph e index, que permitem ao desenvolvedor definir, à sua preferência, as subclasses utilizadas. Assim, para a execução do código dessas classes utilizando o grafo em memória, basta instanciar a classe KL especificando os objetos do grafo e do índice desejados, conforme mostrado na listagem 1, e executar o método executeKL() do objeto kl. A linha 5 armazena o resultado da execução no arquivo kl-mem.out, facilitando ao usuário do *framework* verificar posteriormente seu resultado.

```
1 GraphMem graph = new GraphMem(graphFileName);
2 AbstractPartitionIndex index = new PartitionIndexMem();
3 KL kl = new KL(graph, index);
4 BestPartition resultPartition = kl.executeKL();
5 resultPartition.printPartitions("kl-mem.out");
```

Listagem 1. Uso do *framework* com o grafo em memória

Para a execução do algoritmo KL utilizando o banco Neo4J, basta substituir as linhas 1 e 2 da listagem 1 para instanciar o grafo e o índice apropriados, como mostrado na listagem 2.

```
1 GraphDB graph = new GraphDB(graphFileName);
2 AbstractPartitionIndex index = new PartitionIndexDB(graph);
```

Listagem 2. Substituindo o grafo em memória pelo grafo no banco de dados

Além do *framework* fornecer a implementação dos algoritmos KL (KERNIGHAN; LIN, 1970), FM (FIDUCCIA; MATTHEYSES, 1982), bipartição multinível (KARYPIS; KUMAR, 1995) e *Greedy K-way* (JAIN; SWAMY; BALAJI, 2007) ilustrados na figura 6, ele também auxilia o usuário a implementar outros algoritmos.

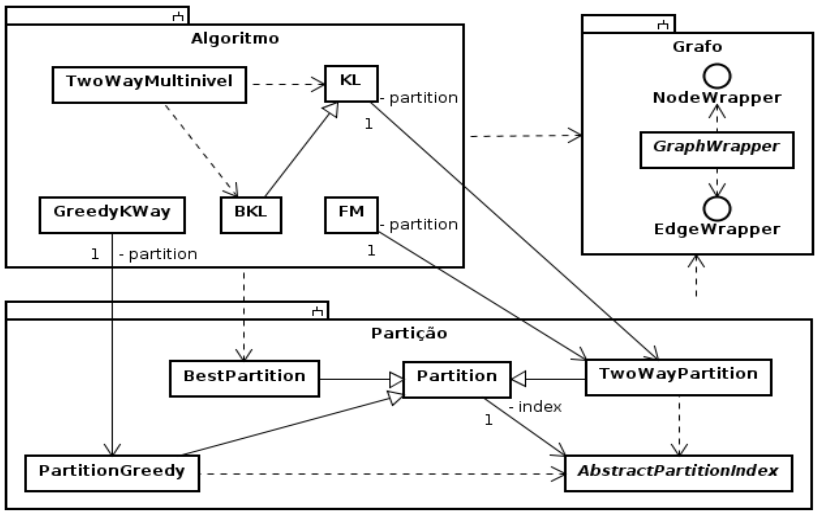

Figura 6. Implementação dos quatro algoritmos utilizando o *framework*

A classe FM implementa os passos de seu algoritmo reaproveitando a classe TwoWayPartition, criada para auxiliar a execução do algoritmo KL.

A classe TwoWayMultinivel executa seu algoritmo utilizando a classe KL para fazer a bipartição no nível mais refinado e a classe BKL, que é uma subclasse de KL, para fazer o refinamento após cada expansão do grafo.

Já a classe GreedyKWay utiliza a classe PartitionGreedy, subclasse de Partition, para armazenar as informações do particionamento.

Finalmente ressalta-se que as classes do pacote "Algoritmo" dependem da classe BestPartition do pacote "Partição" e que tanto as classes do pacote "Partição" quanto as classes do pacote "Algoritmo" dependem das classes e interfaces do pacote "Grafo".

7. Conclusão

O uso deste *framework* facilita a criação de códigos e favorece o aumento de implementações de algoritmos de particionamento de grafos, além de permitir que um mesmo algoritmo seja executado utilizando diferentes formas de armazenamento e representação do grafo processado, pois os pesquisadores envolvidos na área de partireferenciar cite na internet abntcionamento podem realizar seus testes com maior eficiência sem a necessidade de alterações no código do algoritmo para efetuar a mudança da estrutura interna.

O *framework* implementa os componentes propostos permitindo a sua extensão com a implementação do código de outros algoritmos de particionamento ou novas especializações para outros bancos de dados orientados a grafos, o que pode ser facilmente realizado, produzindo, assim, uma biblioteca que forneça suporte para o desenvolvimento de novos e melhores algoritmos de particionamento. Além disso, podem ser criados novos recursos na arquitetura para suportar o particionamento em um ambiente distribuído para o tratamento de grafos dinâmicos, ou seja, que possuem inserções e remoções tanto de vértices quanto de arestas.

Referências

Blondel, V. D., Guillaume, J., Lambiotte, R. e Lefebvre, E. (2008). Fast unfolding of communities in large networks. *Journal of Statistical Mechanics*.

Bonatto, R. S. e Amaral, A. R. S. (2010). Algoritmo heurístico para partição de grafos com aplicação em processamento paralelo. *XLII SBPO*.

Delling, D., Goldberg, A. V., Razenshteyn, I. e Werneck, R. F. F. (2011). Graph partitioning with natural cuts. *IEEE*, 1135–1146.

Delling, D., Görke, R., Schulz, C. e Wagner, D. (2009). Orca reduction and contraction graph clustering. *Springer*, 152–165.

Domingos, D. C., Zaiden, R. F., de Souza, V. J. S., Moreira, E. M. e Carpinteiro, O. A. S. (2012). Sistema de roteirização para transporte público rodoviário. *XLIV SBPO*.

Dutot, A., Olivier, D. e Savin, G. (2011). Centroids: a decentralized approach.

Fiduccia, C. e Mattheyses, R. (1982). A linear-time heuristic for improving network partitions. *Design Automation*, 175–181.

Fiedler, M. (1973). Algebraic connectivity of graphs. *CMJ*, 298–305.

Gehweiler, J. e Meyerhenke, H. (2010). A distributed diffusive heuristic for clustering a virtual P2P supercomputer. *IPDPS Workshops*, IEEE, 1–8.

Hernández, V., Román, J. E., Thomás, A. e Vidal, V. (2007). A Survey of Software for Sparse Eigenvalue Problems. *Technical report*, Universidad Politecnica De Valencia.

Jain, S., Swamy, C. e Balaji, K. (2007). Greedy algorithms for k-way graph partitioning.

Karypis, G. e Kumar, V. (1995). A fast and high quality multilevel scheme for partitioning irregular graphs, Computer Science Department, University of Minnesota, Minneapolis.

Kernighan, B. W. e Lin, S. (1970). An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 291–307.

Nascimento, M. C. V. (2010). Estudo de problemas de particionamento para detecção de comunidades em redes. *XLII SBPO*.

Neo4J, The Neo4j Manual v2.1.8, Neo Technology, 2015 (http://neo4j.com/docs/), 4, 2015.

Newman, M. E. J. e Girvan, M. (2003). Finding and evaluating community structure in networks.

Pellegrini, F. (2007). A parallelisable multi-level banded diffusion scheme for computing balanced partitions with smooth boundaries. *Springer-Verlag*, Nova York, 195–204.

Ruan, J. e Zhang, W. (2007). An efficient spectral algorithm for network community discovery and its applications to biological and social networks. *IEEE Comp. Society*, 643–648.