UNIVERSIDADE FEDERAL DE ITAJUBÁ

PROGRAMA DE PÓS GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

Algoritmos de Particionamento e Banco de Dados Orientado a Grafos

Roberto Ribeiro Rocha

Itajubá, Outubro de 2013

UNIVERSIDADE FEDERAL DE ITAJUBÁ

PROGRAMA DE PÓS GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

Roberto Ribeiro Rocha

Algoritmos de Particionamento e Banco de Dados Orientado a Grafos

Dissertação submetida ao Programa de Pós-Graduação em Ciência e Tecnologia da Computação como parte dos requisitos para obtenção do Título de Mestre em Ciência e Tecnologia da Computação

Área de Concentração: Sistemas de Computação

Orientador: Prof. Dr. Edmilson Marmo Moreira

Coorientador: Prof. Dr. Otávio A. S. Carpinteiro

Outubro de 2013

Itajubá - MG

Ficha catalográfica elaborada pela Biblioteca Mauá
Bibliotecária Jacqueline Rodrigues de Oliveira Balducci- CRB_6/1698

R672a

Rocha, Roberto Ribeiro.
Algoritmos de Particionamento e Banco de Dados Orientado a Grafos. / Roberto Ribeiro Rocha. – Itajubá, (MG) : [s.n.], 2013.
93 p. : il.

Orientador: Prof. Dr. Edmilson Marmo Moreira.
Co-Orientador: Prof. Dr. Otávio Augusto Salgado Carpinteiro.
Dissertação (Mestrado) – Universidade Federal de Itajubá.

1. Grafos. 2. Teoria dos Grafos. 3. Particionamento de grafos. 4. Banco de dados orientado a grafos. I. Moreira, Edmilson Marmo, orient. II. Carpinteiro, Otávio Augusto Salgado, co-orient. III. Universidade Federal de Itajubá. IV. Título.

Agradecimentos

A Deus, pelo dom da vida.

À minha esposa Elizângela e ao meu filho Lucas, pela paciência, confiança e incentivo.

Aos meus pais Ana Maria e Gentil e a meu irmão Ricardo, pelo apoio e suporte.

Ao orientador, por guiar este trabalho.

Ao co-orientador, pelo apoio dedicado.

Aos colegas Emerson, Lênio, Márcio e Ricardo, pelas sugestões, convivência e amizade.

À empresa Liveware, na pessoa de Marcos Okita, que deu suporte para o início desta caminhada.

A FAPEMIG e a CAPES, pelo suporte financeiro.

A todos que ajudaram nessa caminhada.

Resumo

Esta dissertação apresenta uma arquitetura de *software* que permite aos seus usuários implementar algoritmos de particionamento de grafos, possibilitando o reaproveitamento das implementações dos algoritmos em estruturas de armazenamento do grafo em memória ou no banco de dados orientado a grafos Neo4j. Considerando o aumento do volume de informações geradas atualmente, o uso da memória principal se torna um problema, impondo o uso de meios persistentes para o armazenamento das informações através de um banco de dados. Porém, o usuário não deve se preocupar com a forma de armazenamento do grafo, mas sim com a lógica do algoritmo em si, utilizando uma estrutura genérica padronizada. Para dar suporte à elaboração da arquitetura, são apresentados, além dos conceitos de grafos, os aspectos envolvidos no particionamento, que são utilizados pelos algoritmos apresentados, as principais características do banco de dados Neo4J, os diferentes tipos de heurísticas utilizadas, desde o conhecimento local até o uso de técnicas globais de particionamento, com o uso da teoria espectral dos grafos. A arquitetura é validada com a implementação e execução de quatro algoritmos clássicos de particionamento, utilizando grafos sintéticos com corte de arestas conhecidos. Também é mostrado a comparação de desempenho destes algoritmos manipulando grafos maiores disponibilizados pela comunidade.

Palavras-chave: Grafos. Teoria dos grafos. Particionamento de grafos. Banco de dados orientado a grafos.

Abstract

This work presents a software architecture that allows its users to implement graph partition algorithms, enabling the reuse of the algorithms implementation with storage structures of the graph in main memory or at the graph database Neo4J. Considering the increase of the amount of information generated nowadays, the use of main memory turns a trouble, imposing the use of persistent media to store the information through a database. Nevertheless, the user should not worry about the storage form of the graph, but with logic of the algorithm itself, using a generic and standardized structure. In order to support the development of architecture, it is presented, beyond the graph concepts, the aspects involved in the partition process, that are used by the presented algorithms, the main features of the graph database Neo4J, the distinct types of heuristics used, from the local knowledge to the use of global partition techniques, with the use of Spectral Theory of the Graphs. The architecture is validated with the implementation and execution of four classic partition algorithms, using synthetic graphs with known edge cut. It is also shown a performance comparison of these algorithms handing larger graphs provided by the community.

Key-words: Graph. Graph theory. Graph partitioning. Graph database.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

Greedy IIP	*Greedy Iterative Improvement*
ACID	Atomicidade, Consistência, Isolamento e Durabilidade
API	*Application Program Interface*
BFS	*Breath First Search*
BKL	*Boundary Kernighan-Lin*
DiDiC	*Distributed Diffusion Clustering*
FM	*Fiduccia e Mattheyses*
FOS/B	*Firt Order Scheme / Benefits*
FOS/C	*Firt Order Scheme / Constant drain*
FOS/T	*Firt Order Scheme / Truncated*
GPLv3	*GNU Public License version 3*
GSL	*GNU Scientific Library*
KL	*Kerningan-Lin*
NJW	Algoritmo *Ng-Jordan-Weiss*
NoSQL	*Not only SQL*
OoM	*Out of Memory*
Orca	*Orca Reduction and ContrAction Graph Clustering*
SQL	*Structured Query Language*
TEG	Teoria Espectral de Grafos
UML	*Unified Modeling Language*
VLSI	*Very-large-scale integration*
WS	Algoritmo *White-Smyth*
XML	*eXtensible Markup Language*

Lista de Algoritmos

1 Introdução

Muitos dos problemas da vida real de diversos campos do conhecimento, tais como: biologia, física, química, computação, economia, telecomunicações, medicina, etc.; podem ser representados por meio de grafos, através de estruturas de dados representando os elementos desejados, seus atributos e relacionamentos.

A partir do momento que as informações do mundo real são mapeadas para uma estrutura de dados bem definida, elas se tornam propícias para a utilização pelos mais variados tipos de algoritmos, obtendo assim, resultados que representam fenômenos nem sempre esperados por um pesquisador.

Um desses alvos, foco de várias áreas de pesquisa, é a identificação de conjuntos de dados em grupos bem definidos. A partir do momento que os dados são mapeados através de grafos, o agrupamento de informações se torna um problema de **particionamento de grafos**.

O particionamento de grafos permite classificar conjuntos de elementos, representados através de um grafo, levando em consideração certos critérios que identifiquem a fronteira entre estes conjuntos. O critério mais observado na literatura é a minimização do corte de arestas (KERNIGHAN; LIN, 1970), sendo que outras abordagens importantes também podem ser utilizadas para a análise da inter-relação entre os objetos de um determinado conjunto de dados, tais como: balanceamento de vértices (FIDUCCIA; MATTHEYSES, 1982), conectividade (HARTUV; SHAMIR, 2000), centralidade e computação de centróides (DUTOT; OLIVIER; SAVIN, 2011), cortes naturais (DELLING et al., 2011), cobertura (BRANDES; ERLEBACH, 2005), condutância *inter-cluster* (KANNAN; VEMPALA; VETTA, 2004) e medidas qualitativas (NEWMAN; GIRVAN, 2003; BRANDES et al., 2008; ALDECOA; MARÍN, 2011).

Os algoritmos de particionamento são utilizados em várias áreas, como: redes sociais, processamento de imagens, balanceamento de carga em computação para-

lela, biomedicina, mineração de dados, mapeamento genético, etc. e nem sempre fornecem um resultado consistente e adequado, pois estes problemas são complexos, sendo computacionalmente desafiadores (NEWMAN, 2013), exigindo que a busca pela solução seja feita através de heurísticas para se aproximar da solução ótima. Além disso, em muitos casos, o volume de informações a ser processado é elevado, levando a um considerável consumo de recursos computacionais.

Um aspecto relevante na utilização de grafos na solução dos problemas de particionamento é a quantidade de elementos que serão tratados. Os recursos de memória utilizados para manter as estruturas de dados usualmente utilizadas para representar um grafo, possuem limitações. Neste sentido, é essencial o uso de mecanismos de armazenamento persistentes para o tratamento de grandes quantidades de dados.

A persistência das informações de um grafo normalmente é feita através de bancos orientados a grafos, que são otimizados para o armazenamento e busca destas informações estruturadas, com suporte à criação de vértices e arestas, alterações de suas propriedades e fornecimento de mecanismos de consultas baseadas em travessias no grafo (NEO4J, 2013).

Neste contexto, considerando o grande crescimento da quantidade de informações que as aplicações atuais manipulam, é de grande valia uma estrutura de *software* que possibilite ao pesquisador criar, testar e refinar seus algoritmos de forma que utilizem a persistência em disco. Porém, o mesmo algoritmo deve ser capaz de continuar utilizando a memória principal para não perder seus recursos de facilidade de acesso e velocidade.

1.1 Objetivos

O principal objetivo deste trabalho é propor uma solução de arquitetura de *software* para facilitar a implementação de algoritmos de particionamento de grafos, permitindo que sejam executados tanto utilizando estruturas de dados em memória, quanto em banco de dados orientados a grafos. Para alcançar este objetivo, outras metas também foram estabelecidas; são elas:

- apresentar o funcionamento dos principais algoritmos de particionamento de grafos;

- fazer uma análise qualitativa dos algoritmos implementados;
- analisar o comportamento destes algoritmos utilizando um banco orientado a grafos.

Com a intenção de isolar a lógica de um algoritmo de particionamento de sua forma de acesso ao grafo, este trabalho descreve a arquitetura proposta e mostra a facilidade de sua utilização baseada em algoritmos implementados e executados para constatar sua validade.

1.2 Motivação

Os algoritmos criados e melhorados pelos pesquisadores normalmente são implementados utilizando as estruturas de dados em memória principal, pois o foco das pesquisas é conseguir algoritmos que obtenham melhores cortes de arestas em um tempo reduzido.

Como o volume de informações inviabiliza o uso da memória principal, há um aumento na demanda do uso de bancos de dados pelos algoritmos de particionamento. Como os bancos de dados relacionais são inapropriados para armazenar grafos, os bancos orientados a grafos são cada vez mais indicados para esta área.

Diferentemente dos bancos relacionais, que necessitam de *joins* caros para consultar informações, os bancos orientados a grafos armazenam e ligam as referências de acordo com seus registros adjacentes. Essa estrutura diferenciada permite, com certa facilidade, que sejam feitas travessias através do grafo, facilitando a recuperação de informações por um algoritmo de particionamento.

Para um algoritmo beneficiar-se das características de um banco de dados orientado a grafos, é necessário que ele utilize recursos especiais para este acesso, tirando o foco do pesquisador, que está concentrado na heurística e não em como os dados são armazenados.

Isto motiva a elaboração de uma arquitetura de *software* que encapsule esta abstração, padronizando a forma de acesso ao grafo pelo algoritmo, independentemente de seu armazenamento interno, tornando-a uma ferramenta indispensável para atividades deste ramo de pesquisa.

Vale considerar ainda que o particionamento de grafos pode ser aplicado nos mais variados problemas de classificação e agrupamento de informações, onde a

detecção de estruturas de comunidades é importante, pois revelam fenômenos importantes, muitas vezes ocultos.

1.3 Estrutura da dissertação

O trabalho está estruturado em vários capítulos, sendo que no Capítulo 2 são apresentados os conceitos envolvendo o processo de particionamento de grafos, iniciando pela teoria dos grafos e passando pelo problema do particionamento em si com suas diferentes caraterísticas.

O Capítulo 3 aborda o tema "Bancos de dados orientados a grafos", em especial, o Neo4J, que é uma alternativa aos bancos relacionais, facilitando o armazenamento e acesso aos dados, evitando dificuldades existentes em uma modelagem relacional.

O Capítulo 4 contém os algoritmos de particionamento de grafos escolhidos para dar suporte aos requisitos definidos na elaboração deste trabalho.

O Capítulo 5 apresenta a arquitetura de *software* modelada para a implementação dos algoritmos. Ali se concentram os detalhes de modelagem, bem como as soluções encontradas no desenvolvimento do trabalho.

O Capítulo 6 demonstra a utilização da arquitetura desenvolvida e apresenta os resultados experimentais baseados na execução dos algoritmos implementados e, finalizando, o Capítulo 7 apresenta as respectivas conclusões, principais contribuições e sugestões para trabalhos futuros.

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.

Atualmente existem vários tipos de problemas que são mapeados sem muita dificuldade devido à popularidade de aplicações que utilizam estruturas de dados genéricas, bem como novas ferramentas que facilitam a manipulação de informações. Vários algoritmos foram desenvolvidos para que, cada vez mais, fosse possível obter informações importantes a partir dos grafos.

Com o aumento do volume de informações que atualmente é gerado ao redor do mundo, viu-se a necessidade de criar e aplicar novas técnicas para a extração das informações, que são usadas por vários tipos de empresas e instituições para extrair informações de *marketing*, experiências, planejamento entre outras.

Neste contexto, a aplicação de técnicas de particionamento é de grande valia para identificar soluções importantes no que diz respeito à classificação de informações que os grafos representam.

2.1 Conceitos básicos

Um *grafo* $G = (V, E)$ consiste em um conjunto finito V de *vértices* e um conjunto finito E de *arestas* onde cada elemento E possui um par de vértices que estão conectados entre si e pode ou não possuir um peso P.

Um grafo pode ser *direcionado* (*dígrafo*) quando os pares de vértices conectados por uma aresta são ordenados. Em um grafo *não direcionado*, os pares de vértices não são ordenados, assim uma aresta que liga dois vértices u e v pode ser representada tanto como $\{u, v\}$ quanto $\{v, u\}$.

O grafo ainda possui outros atributos, como a *ordem*, que corresponde ao

número de vértices $|V|$ e o *tamanho*, que corresponde ao número de arestas $|E|$.

Um grafo pode ser representado visualmente, com cada vértice sendo um ponto ou um círculo, e cada aresta sendo representada por uma linha que liga os dois vértices aos quais a aresta está associada. A Figura 2.1 ilustra um grafo com 8 vértices e 11 arestas.

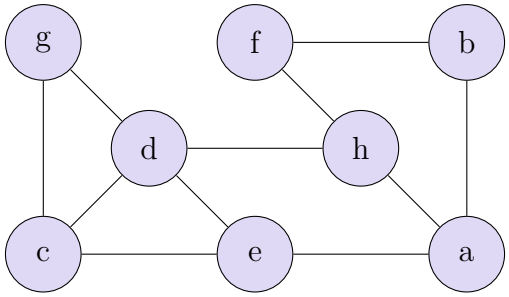


Figura 2.1: Ilustração da representação gráfica de um grafo

A seguir são apresentadas outras definições relacionadas aos grafos (NETO, 1996):

- **grafo simples**: quando o grafo não possui *laços* (arestas que ligam um vértice a si mesmo) ou arestas paralelas, ou seja, duas ou mais arestas que conectam o mesmo par de vértices;
- **grafo completo**: é um grafo simples onde qualquer par de vértices são adjacentes;
- **grafo bipartido**: quando os vértices de um grafo podem ser divididos em dois subconjuntos X e Y tal que toda aresta liga um vértice do subconjunto X a um vértice do subconjunto Y;
- **caminho**: é uma sequência de vértices $\{v_1, \ldots, v_n\}$ que são conectados por arestas $\{e_1 = \{v_1, v_2\}, \ldots, e_m = \{v_{n-1}, v_n\}\}$;
- **grafo conexo**: quando, para qualquer par $\{u, v\}$ dos vértices de um grafo, existe um caminho que liga u e v;
- **subgrafo**: um subgrafo de um grafo G é qualquer grafo H tal que $V(H) \subseteq V(G)$ e $E(H) \subseteq E(G)$;
- **ciclo**: é um caminho onde v_1 e v_n referem-se ao mesmo vértice;

- **componente conectado**: é um conjunto de vértices de um grafo, tal que existe um caminho entre todos os pares de vértices;
- **grau de um vértice**: é a quantidade de arestas conectadas ao vértice.

2.2 Representação computacional de grafos

A necessidade de automatização de processos de manipulação de grafos por meios computacionais exigiu que se criassem diferentes formas de representações através de estruturas de dados, de forma que os algoritmos trabalhassem sobre estas estruturas, extraindo, manipulando e armazenando informações e propriedades dos grafos a fim de chegar aos resultados desejados. Existem várias formas de representar um grafo computacionalmente, como mostrado a seguir:

- **Matriz de adjacência**: é uma matriz $n \times n$

$$\mathbf{A}_G \leftarrow (a_{uv})$$

onde a_{uv} é 1 se existe uma aresta ligando os vértices u e v, isto é, se os vértices em questão são adjacentes. Caso contrário, a_{uv} possui o valor 0. Quando as arestas possuirem um peso ou custo, o valor de a_{uv} deve refletir o peso correspondente da aresta. A matriz da Figura 2.2 é um exemplo deste tipo de representação, correspondendo ao grafo da Figura 2.1.

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

Figura 2.2: Matriz de adjacência do grafo da Figura 2.1

- **Lista de adjacência**: nesta representação, cada vértice v possui uma lista, de vértices vizinhos, que são adjacentes ao vértice em questão, que, em alguns

casos especiais onde a quantidade de arestas é reduzida, consome menos memória do que as matrizes mencionadas anteriormente. A Figura 2.3 mostra um exemplo desse tipo de representação.

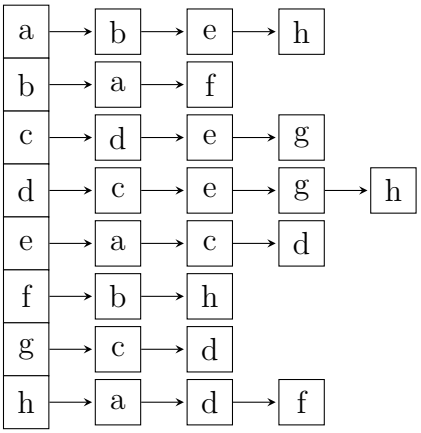


Figura 2.3: Lista de adjacências do grafo da Figura 2.1

- **Objetos**: neste tipo de representação, mostrado na Figura 2.4, são utilizados os recursos das linguagens orientadas a objetos para organizar os vértices e arestas para montar o grafo desejado (mais detalhes serão discutidos na seção 3.2).

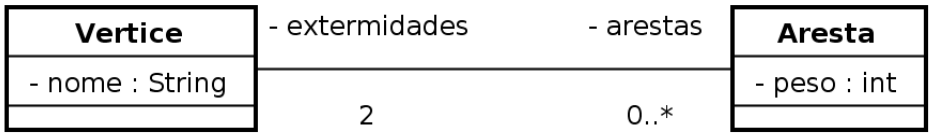


Figura 2.4: Classes representando vértices e arestas

Os atributos das classes são utilizados para armazenar os relacionamentos entre os objetos envolvidos, da seguinte maneira:

  - um objeto **vértice** possui uma lista de objetos aresta. Isso permite o acesso fácil e rápido do vértice atual para qualquer uma de suas arestas.
  - um objeto **aresta** possui dois atributos do tipo vértice, um indicando a ponta a e outro indicando a ponta b. Esses dois objetos facilitam a navegação, pois conhecendo o objeto em uma extremidade da aresta, facilmente identifica-se a outra extremidade, permitindo assim que a navegação seja feita através desta aresta. Essa aresta também pode possuir um atributo indicando seu peso.

2.3 O problema do particionamento

A necessidade de classificação de elementos em conjuntos distintos e da identificação das características comuns de cada conjunto levou a várias soluções onde, após o mapeamento do problema para um grafo, tornou-se possível o tratamento e manipulação das informações pelo particionamento de grafos.

Segundo Kernighan e Lin (1970), o problema do particionamento de um grafo $G = (V, E)$ consiste em dividir esse grafo em k subconjuntos de vértices de maneira que o corte de arestas seja minimizado e que cada subconjunto possua uma quantidade máxima de vértices.

O **corte de arestas** corresponde o conjunto de arestas nas quais seus vértices estejam em diferentes partições, e seu valor é dado pela soma de seus pesos. Para grafos cujas arestas não possuam peso, o peso é considerado unitário.

Existem algumas classificações nas quais o particionamento se enquadra, dependendo de suas características. Quanto à quantidade de partições, existem dois tipos distintos, como mostrado a seguir:

- **bipartição**: corresponde à divisão do grafo em apenas dois conjuntos de vértices, podendo possuir quantidades semelhantes de vértices em cada conjunto;
- **particionamento *k-way***: corresponde ao processo de divisão do grafo em k conjuntos de vértices, e cada conjunto possuindo uma quantidade próxima de $|V|/k$ vértices. Uma das técnicas utilizadas nesta abordagem é aplicar a bipartição recursivamente, porém há a restrição da quantidade de partições obtidas.

Quanto à heurística de busca de soluções, existem várias abordagens expostas por Fortunato (2010), cada uma com suas características:

- **Locais**: algoritmos desta família fazem a busca da solução utilizando os vértices vizinhos aos vértices que estão sendo processados em um dado momento. Os algoritmos locais são mais difundidos e possuem uma gama maior de variações e técnicas para o particionamento. São eles:
  - **gulosos**: cria conjuntos de vértices que são iniciados a partir de vértices

aleatórios e utilizam técnicas gulosas para avançar no grafo atribuindo apropriadamente cada vértice a um conjunto;

- **divisivos**: utilizam do recurso de remoção de arestas chave, que, após várias remoções, permite desconectar o grafo, obtendo, assim, as partições desejadas;
- **aglomerativos**: possuem a característica de agrupar vértices considerados próximos de forma que o melhor candidato é incorporado ao conjunto apropriado. A cada iteração, os vértices são aglomerados e contraídos em um único vértice, reduzindo assim a quantidade de vértices, até chegar na quantidade de conjuntos desejada;
- **difusivos**: utilizam técnicas de difusão de líquido ou gases, de forma que, a partir de vértices iniciais, diferentes líquidos são injetados em cada vértice e em cada iteração esses líquidos são transferidos para outros vértices até se encontrarem e se anularem, formando uma fronteira suave que indica o corte.

- **Globais**: são algoritmos que utilizam as características da matriz de um grafo para alcançar o particionamento. A técnica estudada neste trabalho para o particionamento global utiliza a teoria espectral dos grafos, detalhada na seção 2.3.2.

- **Multiníveis**: são métodos que utilizam de contração de vértices através de emparelhamento, contraindo o grafo original para obter um grafo significativamente menor. Após o particionamento do grafo contraído, basta repassar o particionamento feito até chegar ao grafo em seu ponto original.

- **Métodos usando otimização**: são métodos que usam recursos de inteligência artificial, algoritmos genéticos (MENÉNDEZ; CAMACHO, 2012), *simulated anneling* (SCHAEFFER, 2005) e simulação de Monte Carlo para aproximarem de uma boa solução de particionamento.

- **Métodos mistos**: podem utilizar da combinação dos métodos anteriores aproveitando as melhores características de cada um para definir um novo método.

Além desta classificação, existem algoritmos que utilizam informações de coordenadas dos vértices para efetuar o particionamento, os quais não são abordados por este trabalho.

A subseção seguinte apresenta vários conceitos utilizados pelos vários algoritmos estudados nessa dissertação.

2.3.1 Conceitos envolvidos no particionamento

Com o surgimento dos algoritmos de particionamento, vários conceitos foram criados para suportar as heurísticas de execução a fim de melhorar os algoritmos já existentes e até influenciando a criação de novas formas de solução deste problema. Vários destes conceitos são facilmente representados matematicamente e permitem entender melhor a intenção de cada algoritmo.

Os primeiros conceitos levantados por Kernighan e Lin (1970) foram o corte mínimo e o ganho, onde o **corte mínimo** é o objetivo do processo de minimização do corte de arestas. O **ganho** de um vértice indica a soma dos pesos das arestas, na qual será diminuído do corte de arestas, caso ele seja movido de uma partição para outra.

A **contração de arestas**, que será discutida na seção 4.3, faz o emparelhamento de vértices, gerando um novo vértice a partir de dois vértices anteriores, produzindo um grafo menor que o original e permitindo a execução de um algoritmo de particionamento com este grafo menor.

Por outro lado, a **expansão de arestas** consiste em obter os dois vértices originais contidos em um vértice contraído, fazendo o caminho de volta do grafo contraído para o grafo original.

Outros conceitos importantes são:

- **modularidade**: definida por Newman e Girvan (2003) como sendo uma medida de qualidade de um particionamento de um grafo em k comunidades. Ela mede a densidade de arestas dentro das partições comparado à quantidade de arestas entre as partições e é definida como

$$Q = \sum i(e_{ii} - {a_i}^2) \tag{2.1}$$

  onde e_{ii} corresponde à quantidade de arestas cujos vértices pertencem à partição i e a_i é a quantidade total de arestas que possui, pelo menos, um vértice pertencente à partição i;

- **centralidade**: foi discutida por Freeman (1979), indicando uma posição especial de um vértice em relação a seus vizinhos, e se baseia em três conceitos:

    1. grau: identifica vértices com uma grande concentração de vizinhos;
    2. *betweenness*: indica a frequência em que um vértice pertence entre pares de outros vértices no menor caminho entre eles;
    3. *closeness*: indica a proximidade de um vértice em relação aos demais vértices no grafo.

2.3.2 Teoria espectral dos grafos

A Teoria Espectral de Grafos (TEG) é uma área da Matemática Discreta e da Álgebra Linear que estuda as propriedades de um grafo a partir das informações fornecidas pelo seu espectro (HOGBEN, 2005).

O **espectro de um grafo** se caracteriza pelo relacionamento entre as propriedades algébricas do espectro de certas matrizes associadas a um grafo e as propriedades topológicas desse grafo. A obtenção dos espectros mais utilizados, ou seja, as associações mais comuns, são as feitas através da matriz de adjacência e da matriz laplaciana. O espectro da matriz laplaciana de um grafo é chamado de **espectro do laplaciano** (ABREU, 2011).

A matriz laplaciana $L(G)$ de um grafo G é definida como:

$$L = D - A \tag{2.2}$$

onde D é a matriz diagonal dos graus dos vértices do grafo G, ou seja, a matriz tal que $D_{ii} = d(v_i)$ e A é a matriz de adjacência de G.

O espectro de um grafo possui várias informações sobre ele, sendo que, para certas famílias de grafos, é possível caracterizar um grafo através de seu espectro. Porém, na maioria dos casos, isso não é possível, mas mesmo assim o espectro possui informações úteis sobre o grafo.

Dentre as várias propriedades obtidas através do espectro de um grafo, cita-se a quantidade de subgrafos elementares de G com i vértices decorre do $i-$ésimo coeficiente de seu polinômio característico (GODSIL; ROYLE, 2001). Merris (1994) também apresenta vários exemplos onde, através do espectro de um grafo, pode-se caracterizar vários deles através de classes, como os ciclos, grafos completos e

bipartidos.

O espectro de um grafo é o conjunto de autovalores λ, normalmente apresentados em ordem decrescente, associados às suas respectivas multiplicidades algébricas. Cada autovalor λ possui seu respectivo autovetor associado.

O cálculo dos autovalores e autovetores consiste em resolver o seguinte sistema de n equações lineares

$$Ax = \lambda x \tag{2.3}$$

onde $A = [a_{jk}]$ é uma matriz $n \times n$ do grafo, λ é um escalar e x é um autovetor associado a λ.

Assim, um valor de λ tal que $x \neq 0$ seja uma solução do sistema é chamado de **autovalor** ou **valor característico** da matriz A. As correspondentes soluções $x \neq 0$ são chamadas de **autovetores** ou **vetores característicos** associados ao autovalor λ.

O processo de solução consiste em encontrar as raízes do *polinômio característico* de A, dada pela equação polinomial p de grau n na variável λ

$$pA(\lambda) = det(A - \lambda I) = 0 \tag{2.4}$$

onde I é a matriz identidade e suas n raízes são seus respectivos autovalores. A multiplicidade algébrica de λ é o número de vezes que λ ocorre como raiz do polinômio $pA(\lambda)$.

Com os autovalores determinados, os respectivos autovetores são obtidos resolvendo o sistema de equações lineares correspondente da matriz.

Dado que a expansão direta do determinante da equação 2.4 para a determinação do polinômio característico é ineficiente e não trivial para matrizes de grandes ordens, existem vários métodos numéricos e iterativos para evitar esse cálculo de determinante. Hernández et al. (2007) citam alguns métodos para solução de problemas de autovalores com matrizes esparsas:

- método iterativo *Single and Multiple Vector*;
- método de *Arnoldi*;

- método de *Lanczos*;
- *Singular Value Decomposition*;
- método de *Davidson and Jacobi-Davidson*;
- método de *Optimization and Preconditioned*.

Existem algumas bibliotecas de *software* nas linguagens C++ e Java já implementadas que fazem o cálculo de autovalores e autovetores. Em C++, a mais famosa delas é a *GNU Scientific Library* (GSL) (CONTRIBUTORS, 2010). Dentre as bibliotecas em Java pode-se citar: Jama (HICKLIN et al., 2013), JLinAlg (KEILHAUER et al., 2013) e Jlapack (DONGARRA; DOWNEY; SEYMOUR, 2013).

Uma outra característica importante para o particionamento é a quantidade de *clusters* que o espectro de um grafo permite conhecer aproximadamente (LUXBURG, 2007), obtido através do valor de k, tal que, os k autovalores λ_1, λ_2, ..., λ_k possuem valores muito pequenos, sendo que o autovalor λ_{k+1} possui um valor relativamente grande em relação aos anteriores.

2.3.3 Particionamento espectral

O espectro de um grafo, feito a partir de sua matriz laplaciana, possui um destaque especial devido ao seu segundo menor autovalor, conhecido por **conectividade algébrica** $a(G)$, que fornece informações sobre a conectividade de um grafo, e com isso pode ser aplicado ao particionamento de grafos (FIEDLER, 1973).

O autovetor correspondente ao segundo menor autovalor λ_2 é chamado de **vetor de Fiedler**, pois ele contém informações que dividem um grafo em dois conjuntos, A e B, tal que o corte de arestas entre esses conjuntos é o mínimo possível.

Dessa forma, este tipo de particionamento, conhecido como método de *Fiedler*, é uma área de grande interesse tanto na Álgebra quanto no particionamento de grafos.

2.4 Considerações finais

A utilização de grafos para representar elementos e seus relacionamentos do mundo real é uma boa estratégia de transportar estas informações para o ambiente computacional, seja através das estruturas de dados. Esta transformação, porém, deve ser cuidadosa, pois se mal definida pode inviabilizar o processo para a solução.

Os algoritmos de particionamento de grafos são heurísticas para encontrar a melhor forma de agrupamento entre conjuntos de dados representados por um grafo, tendo como principal objetivo minimizar o corte de arestas. As abordagens utilizadas no particionamento podem ser locais ou globais. As locais buscam a solução através da vizinhança dos vértices enquanto que as globais utilizam normalmente o espectro da matriz que representa o grafo.

Outros conceitos e características de particionamento de grafos podem ser vistos em Fortunato (2010) e outras propriedades espectrais são mostradas por Luxburg (2007) e Nascimento e Carvalho (2011).

3 Banco de Dados Orientado a Grafos

Um sistema gerenciador de banco de dados é uma ferramenta muito importante para a computação, pois se encarrega de todo o processo de gerenciamento e armazenamento de dados de forma organizada, possuindo ferramentas para facilitar e acelerar a busca de informações que foram previamente armazenadas.

Atualmente a grande maioria dos bancos de dados é do tipo **relacional**, no qual as informações estão armazenadas em tabelas e associadas através de relacionamentos, permitindo uma fácil organização e obtenção de informações.

Com o aumento do volume de informações gerado nos ambientes reais e mesmo simulados, também há uma demanda no processamento de transações relativas à busca e atualização de dados nos bancos de dados. Porém, existem vários ambientes do mundo real que necessitam ser adaptados ao modelo relacional, que é o utilizado pelos bancos tradicionais, para que se consiga utilizar seus recursos de forma eficiente.

3.1 Neo4J

Frente a esse problema de adaptação, foram criados outros tipos de bancos de dados chamados de *NoSQL*, um acrônimo para *Not only SQL*, indicando que esses bancos não usam somente o recurso de *Structured Query Language* (SQL), mas outros recursos que auxiliam no armazenamento e na busca de dados em um banco não relacional. Vários modelos foram implementados, por exemplo, bancos orientados a documentos, bancos *eXtensible Markup Language* (XML) e bancos orientados a grafos. Esse último está tendo uma grande expansão devido às aplicações voltadas a redes sociais que estão sendo utilizadas por boa parte da população e para pesquisas nas áreas biológicas, processamento de imagens, bioquímica entre outras.

Os bancos de dados orientados a grafos possuem algumas características que os diferem dos bancos relacionais, no que diz respeito à forma de armazenamento e busca. Eles armazenam diretamente os vértices e arestas sem o uso de tabelas, permitindo a execução de consultas rápidas através de **travessias** no grafo, acessando somente os vértices pertencentes àquele escopo da consulta, evitando *joins* caros, muito utilizados nos bancos relacionais (ROBINSON; WEBBER; EIFREM, 2013).

Neste trabalho, foram estudadas algumas características do banco de dados orientado a grafos Neo4J (NEO4J, 2013), que é um banco *NoSQL*, sob a licença *GNU Public License version* 3 (GPLv3). Ele possui alta disponibilidade e é escalável a bilhões de vértices e arestas, oferecendo recurso de *query* através de um *framework* chamado ***traversal***, que navega no grafo para obter as informações desejadas. Ele ainda pode ser instalado em um ambiente multisservidor ou pode ser executado embarcado em um programa Java, que é o caso da implementação deste trabalho.

O Neo4J ainda possui suporte à transações com atomicidade, consistência, isolamento e durabilidade (ACID) e possui o *framework Cypher Query Language* que permite escrever consultas através de uma linguagem formal, porém de forma fácil para um ser humano entendê-la. O Neo4J também já possui alguns algoritmos implementados como por exemplo: *Shortest Path*, *All Simple Paths*, *All Paths*, *Dijkstra* e *A**.

Os vértices e arestas podem ser indexados de forma prática e eficiente graças ao uso do *framework* **Lucene** (NEO4J, 2013).

3.2 Elementos de um banco de dados orientado a grafos

Um banco orientado a grafos possui vários elementos, mostrados na Figura 3.1, nos quais são implementados toda a estrutura de dados e algoritmos internos.

O banco de dados gerencia os índices e o grafo em si. O grafo armazena as informações em vértices e arestas (relacionamentos) que são mapeados por índices a partir das propriedades de cada um. Os relacionamentos 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. Por outro lado, existem as travessias que navegam no grafo para identificar

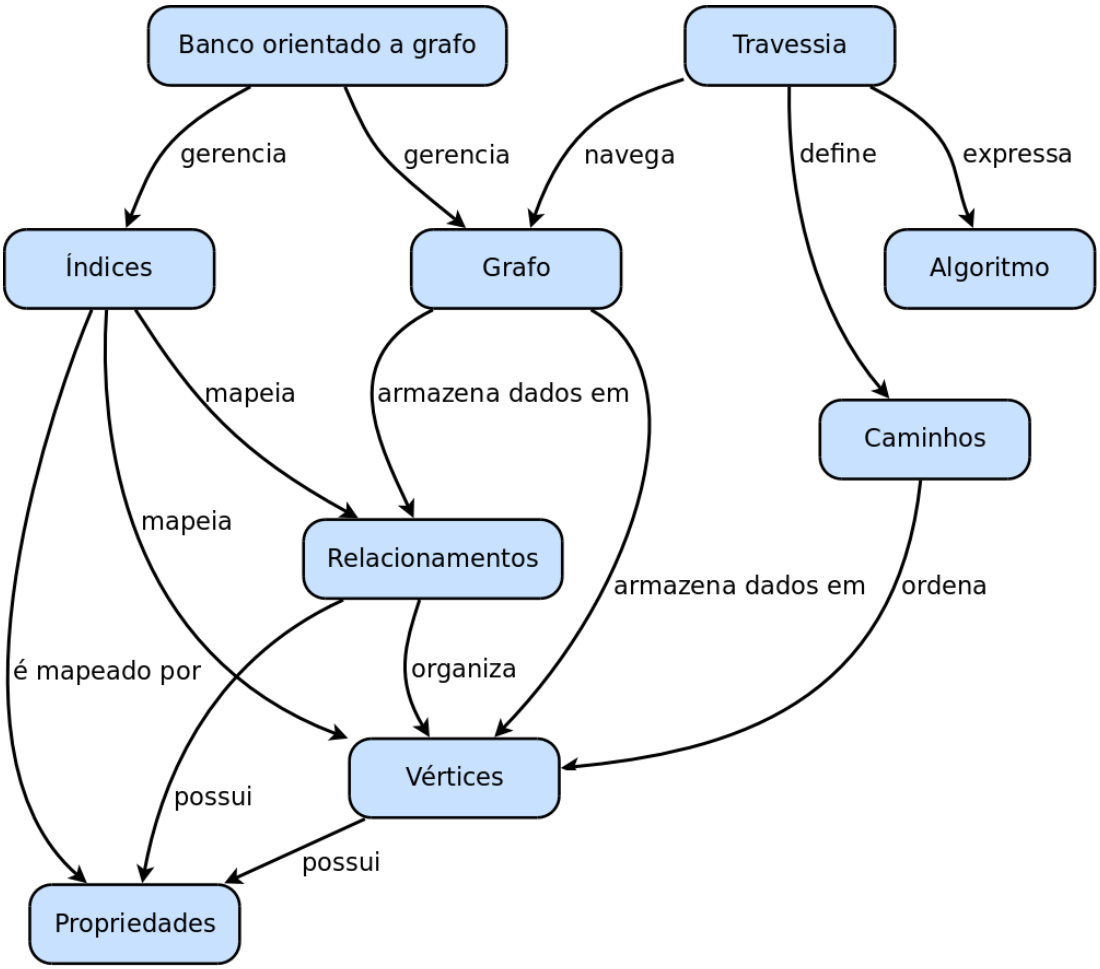


Figura 3.1: Visão geral: banco de dados orientado a grafo, adaptado de Neo4J (2013)

caminhos, executando assim algum algoritmo (NEO4J, 2013).

Conforme ilustra a Figura 3.2, os relacionamentos possuem um vértice inicial e um final juntamente com seu tipo, permitindo a existência de mais de um relacionamento, com tipo diferente, entre o mesmo par de vértices indicando diferentes relações entre eles. Estes recursos possibilitam fazer a navegação pelos vértices do grafo, diferenciando diferentes tipos de arestas, podendo assim, implementar o algoritmo desejado.

A Figura 3.3 apresenta a estrutura de uma propriedade de um vértice ou de um relacionamento, na qual possui uma chave e um valor associado, permitindo armazenar informações úteis sobre cada elemento do grafo.

Para este trabalho, foi utilizado o Neo4J, de forma embarcada, na implementação da arquitetura, facilitando assim a execução dos testes e tornando o ambiente independente de uma configuração externa.

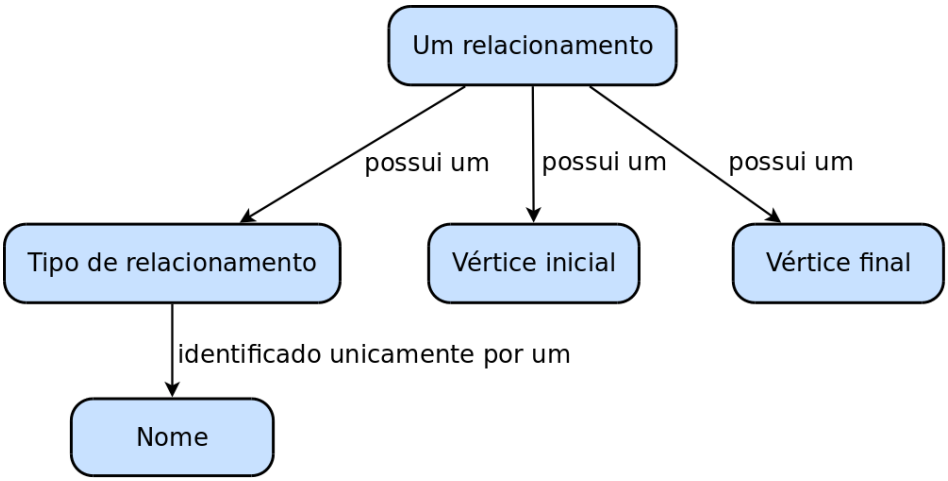


Figura 3.2: Estrutura de um relacionamento, adaptado de Neo4J (2013)

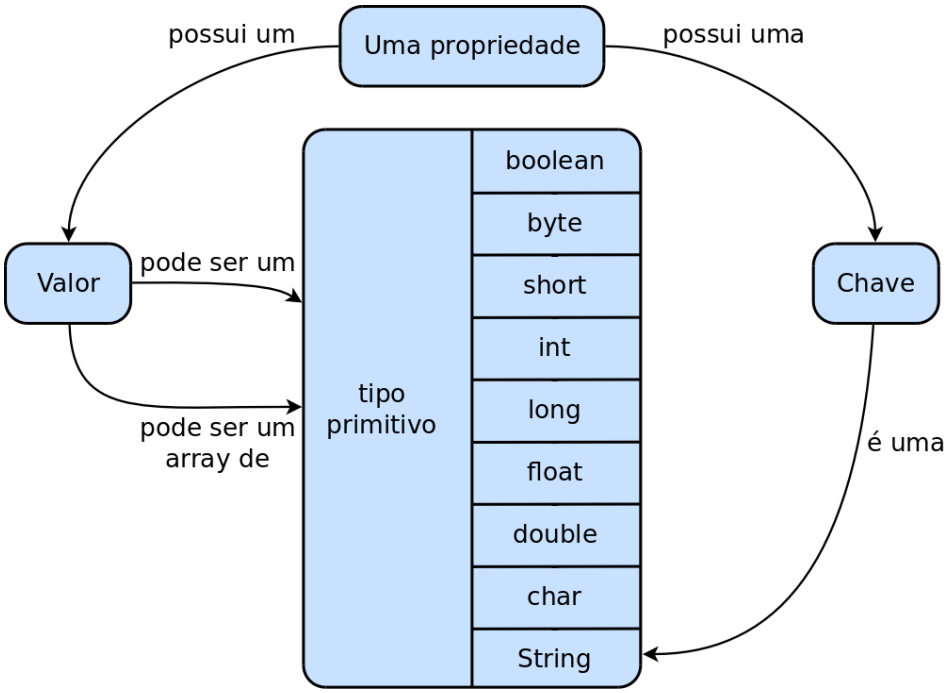


Figura 3.3: Propriedades de vértices e arestas, adaptado de Neo4J (2013)

Para usufruir dessa ferramenta, bem como sua *Application Program Interface* (API), foi necessário adicionar a dependência Maven do Neo4J no *build path* do projeto Java e utilizar as interfaces e classes, das quais as mais importantes são (NEO4J, 2013):

- ***GraphDatabaseService***: interface que provê o ponto de acesso principal para uma instância do Neo4J, definindo serviços básicos de criação do banco e de vértices, recuperação de vértices e arestas entre outros;
- ***EmbeddedGraphDatabase***: implementação de *GraphDatabaseService* para

uso embutido em um programa Java, permitindo a criação e uso do banco em um diretório local;

- ***Index***: interface que define os serviços de criação e uso de índices baseados em pares chave e valor, que podem ser criados tanto para vértices quanto para arestas;
- ***RelationshipType***: interface para definir o tipo do relacionamento que foi criado entre dois vértices. Essa abordagem permite que dois vértices possuam mais de um relacionamento, porém com tipos diferentes, indicando diferentes interações entre eles;
- ***Transaction***: interface que permite o manuseio de transações por meio de programação;
- ***PropertyContainer***: define uma API para trabalhar com propriedades dos vértices e arestas;
- ***Node***: interface que representa o vértice, possuindo métodos para criação e recuperação de arestas e métodos úteis de travessias;
- ***Relationship***: interface que representa a aresta, contendo seu tipo e dois objetos representando os vértices (*startNode* e *endNode*);
- ***GlobalGraphOperations***: classe que fornece serviços de operações globais no banco, como recuperar todos os vértices e todas as arestas.

As interfaces *Node* e *Relationship* são as mais usadas, pois seus objetos são os que realmente contém as informações manipuladas e armazenadas pelo grafo. Elas foram definidas de forma genérica, conforme ilustra o diagrama da Figura 3.4.

De acordo com a documentação do Neo4J, a interface `PropertyContainer` define os serviços para manipulação das propriedades, através de três métodos: `getProperty()`, `setProperty(...)` e `hasProperty(...)`, que estão associadas diretamente com os objetos vértices ou arestas. A chave e o valor de cada propriedade é definida de acordo com a necessidade do usuário, possibilitando tanto aos vértices quanto às arestas armazenarem as informações pertinentes ao grafo, tornando esta estrutura flexível (NEO4J, 2013).

A interface `Node` define os métodos para recuperar seu `id` interno do banco, verificar se ele possui arestas e recuperar as arestas que estão conectadas a ele. Já

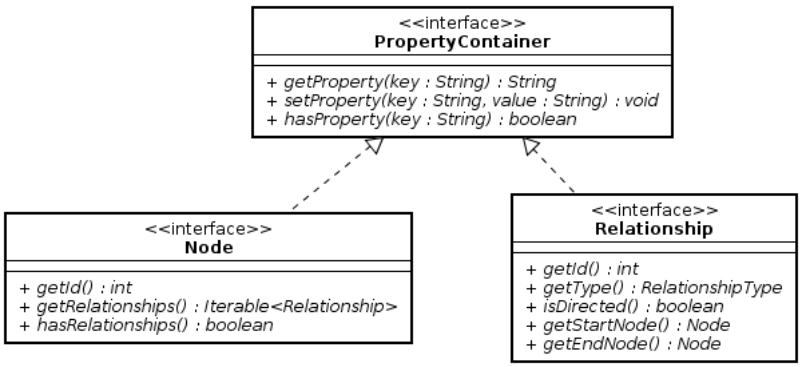


Figura 3.4: Interfaces *Node* e *Relationship*, adaptado de Neo4J (2013)

com a interface `Relationship` é possível recuperar o `id` e seu tipo, definido por `RelationshipType`, verificar se ela é direcionada ou não e recuperar os vértices de suas extremidades, através dos métodos `getStartNode()` e `getEndNode()`.

3.3 Considerações finais

Estas interfaces e classes foram utilizadas na definição da arquitetura, detalhada no Capítulo 5.

A utilização de um banco de dados orientado a grafos, além de evitar que a memória seja utilizada para armazenar o grafo, permite que seja realizado o particionamento, a partir de um grafo já existente no banco, armazenando o resultado nas propriedades dos próprios vértices, e após o particionamento, o grafo pode continuar a ser utilizado por outras aplicações.

4 Algoritmos de Particionamento

Os algoritmos de particionamento de grafos surgiram da necessidade de se resolver problemas do mundo real para agrupamento de elementos. Basicamente, iniciado por Kernighan e Lin (1970) para colocar componentes eletrônicos em placas de circuitos impressos com o objetivo de minimizar o número de conexões entre as placas.

Estes algoritmos, então, passaram a ser alvos de estudo por outros pesquisadores, atingindo outros ramos da ciência, de forma a resolver os mais variados tipos de problemas referentes a particionamento e agrupamento.

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 de várias formas uma solução próxima da ótima. Muitas vezes, essa boa solução consiste em um mínimo local alcançado pela heurística. Melhorias nessa boa solução implicam em consumir mais tempo de processamento ou mais recursos, como a memória, através de aperfeiçoamentos nos algoritmos. Muitas vezes, essas melhorias não compensam o tempo ou o consumo de memória, mantendo assim a solução quase ótima alcançada anteriormente.

Este capítulo, menciona os algoritmos de particionamento de grafos utilizados neste trabalho, bem como, alguns exemplos para simplificar seu entendimento.

4.1 *Kerningan-Lin* (KL)

O primeiro algoritmo heurístico voltado para o particionamento de grafos foi o proposto por Kernighan e Lin (1970) no qual um dos objetivos da época era dispor, da melhor maneira possível, os componentes em uma placa de circuitos impressos, chamados de *Very-large-scale integration* (*VLSI*), de forma que os componentes altamente conectados ficassem na mesma placa enquanto que componentes fra-

camente conectados poderiam ficar em placas distintas, minimizando as ligações entre placas.

Seu funcionamento, mostrado no Algoritmo 1, considera encontrar um particionamento admissível de um grafo G com um custo mínimo, usando uma heurística para encontrar boas soluções ao invés de usar métodos exaustivos para tentar achar uma solução ótima, sendo que, em muitos casos, boas soluções são mais valiosas que soluções ótimas, devido à grande quantidade de tempo necessário para encontrar estas últimas.

Algoritmo 1: Algoritmo KL (KERNIGHAN; LIN, 1970)

```
Entrada: O grafo G
Saída: O particionamento P
Atribuir aleatoriamente os vértices às duas partições de P
repita
    Computar o valor de D para todos os pares de vértices
    repita
        Calcular o ganho para cada par de vértices
        Selecionar o par com maior ganho e excluí-lo dos próximos cálculos
        Recalcular o valor de D para os pares restantes
    até processar todos os vértices
    Escolher os k pares para maximizar G
    Fazer a troca dos k pares na partição P
até G ≤ 0
retorna P
```

A intensão do algoritmo é criar dois conjuntos arbitrários A e B a partir de G, como particionamento inicial, e tentar diminuir o custo externo inicial T por uma série de trocas de subconjuntos de A e B. Segundo Kernighan e Lin (1970), quando não for possível fazer mais melhorias, o particionamento resultante A' e B' será mínimo localmente com uma boa probabilidade de ser um mínimo global.

Esse processo pode então ser repetido com a geração de uma outra partição inicial arbitrária A e B e assim por diante, para obter o número de particionamentos desejado com diferentes mínimos locais.

A ideia dessa heurística é identificar um elemento de cada lado da partição (*2-way*) de forma que ao fazer a troca desses elementos, isto é, o elemento x de A passa a pertencer ao conjunto B e o elemento y de B passa a pertencer ao conjunto A, o custo do corte seja reduzido, sem considerar todas as trocas possíveis. Assim, a principal questão se concentra em fazer a escolha adequada desses elementos.

O cerne da execução do algoritmo consiste em identificar um par de vértices, de forma aproximada, calculando para cada vértice v um custo externo E e um custo interno I, da seguinte forma:

$$E(v) = \sum_{y \in B} c_{v,y} \tag{4.1}$$

$$I(v) = \sum_{x \in A} c_{v,x} \tag{4.2}$$

onde $c_{v,y}$ é o peso(custo) da aresta que conecta o vértice v a um vértice do conjunto B e $c_{v,x}$ é o peso da aresta que conecta o vértice v a um vértice do conjunto A.

Após o cálculo dos custos, é calculado a diferença entre os custos externo e interno de cada vértice de ambas partições:

$$D(v) = E(v) - I(v) \tag{4.3}$$

Para decidir qual par de vértices $va \in A$ e $vb \in B$ deve ser trocado entre as partições, calcula-se o fator do ganho, ou seja, a redução do custo, para os pares de vértices, expresso por:

$$g(a,b) = D(va) + D(vb) - 2c(va, vb) \tag{4.4}$$

onde c é o peso da aresta entre va e vb. O fator g pode ser tanto positivo quanto negativo. Valores negativos podem indicar que a troca dos vértices fará com que a solução escape de um mínimo local, melhorando o resultado final do particionamento.

O próximo passo é identificar o par que produz o maior ganho e armazená-lo temporariamente. Então, o algoritmo recalcula os valores de D para os elementos que ainda não foram processados na iteração atual, ou seja, todos os elementos $r_1 \in \{A - (a)\}$ e $s_1 \in \{B - (b)\}$. Esse cálculo é dado por

$$D'(r) = D(r) + 2c(r,a) - 2c(r,b) \tag{4.5}$$

$$D'(s) = D(s) + 2c(s,b) - 2c(s,a) \tag{4.6}$$

O algoritmo volta a calcular o ganho para os pares de vértices restantes, escolhendo um novo par e recalculando D, até todos os vértices terem sido analisados.

Ao final do cálculo do ganho de todos os pares, escolhe-se k pares para maximizar a soma parcial G dos ganhos calculados

$$G = \sum_{i=1}^{k} g_i \tag{4.7}$$

Se $G > 0$, então uma redução no custo com o valor de G pode ser feita trocando os k pares. Após essa troca, o particionamento resultante é tratado como a partição inicial para um novo processamento ou pode ser retornado como uma solução final.

4.2 *Fiduccia e Mattheyses* (FM)

Essa foi a segunda heurística desta área de grafos, proposto por Fiduccia e Mattheyses (1982), que visa melhorar de modo iterativo um particionamento.

A ideia básica, mostrada no Algoritmo 2, consiste em, a partir de um particionamento inicial, mover um vértice por vez de um bloco para outro a fim de minimizar o tamanho do corte, baseado em um critério de balanceamento das partições.

Algoritmo 2: Algoritmo FM (FIDUCCIA; MATTHEYSES, 1982)

Entrada: O grafo G, a partição inicial P
Saída: A partição P

```
Computar o valor do ganho para cada vértice de G
repita
    Selecionar um vértice c_i com maior ganho e que satisfaça o critério de
    balanceamento
    se não encontrou um vértice então
        interromper loop
    Remover c_i dos próximos cálculos
    Atualizar os ganhos dos vértices vizinhos de c_i
até processar todos os vértices de G
Escolher os k vértices c_1, ..., c_k para maximizar G
se G > 0 então
    Fazer a troca de partição dos k vértices
retorna P
```

O cálculo do balanceamento é feito a partir da soma dos pesos dos vértices de cada partição e é dado por:

$$r \cdot V - S_{max} \leq A \leq r \cdot V + S_{max} \tag{4.8}$$

onde r é o fator de balanceamento, normalmente $0 < r < 1$, para permitir uma certa flexibilidade no movimento dos vértices, A é a soma dos pesos de todos os vértices da partição A, V é a soma dos pesos de todos os vértices do grafo e S_{max} é o maior peso de um vértice do grafo.

A escolha do vértice a ser movido, chamado de *vértice base*, é baseada no ganho $g(i)$ do vértice i, indicado, como exemplo, pelo valor dentro de cada vértice na Figura 4.1, como sendo o número de arestas pelo qual o corte diminuiria, devido à mudança de partição deste vértice. O critério de balanceamento evita que todos os vértices migrem de um bloco para outro. Mesmo se o ganho não for positivo, o vértice é movido, com a expectativa que o movimento permita que o algoritmo **saia de um mínimo local**.

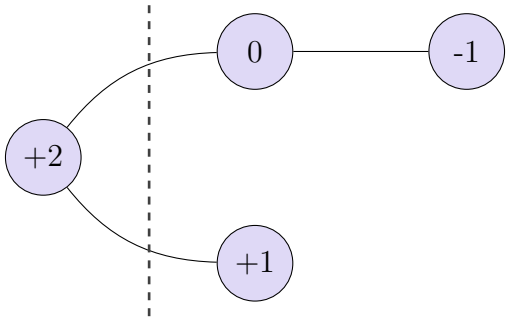


Figura 4.1: Exemplo de ganho dos vértices (FIDUCCIA; MATTHEYSES, 1982)

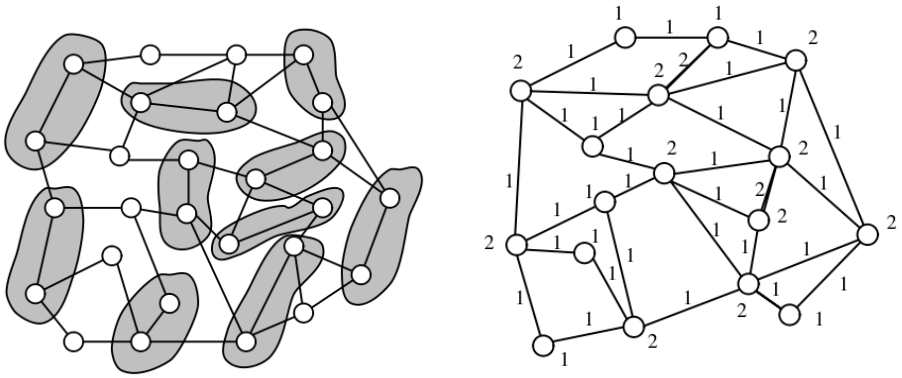
Após feitos todos os movimentos, o melhor particionamento encontrado durante o passo é utilizado como saída do passo. Essa técnica de minimização é herdada de Kernighan e Lin (1970).

Os vértices já movidos são marcados, para evitar a migração de uma partição para outra indefinidamente, sendo que somente vértices livres podem fazer um movimento em cada passo do algoritmo. Esta marcação também permite que se possa designar certos vértices **fixos** em uma partição, permitindo que o algoritmo refina partições criadas por execuções anteriores.

4.3 Bipartição multinível

A principal ideia do algoritmo proposto por Karypis e Kumar (1995) é contrair o grafo original, obtendo um grafo equivalente reduzido, para minimizar o esforço de particionamento, executando três fases bem definidas.

1. **Contração**: Durante a fase de contração o grafo original G_0 é transformado em uma sequência de grafos menores G_1, G_2, ..., G_n, cada um com menos vértices, tal que $|V_0| > |V_1| > \ldots > |V_n|$, porém preservando as propriedades do grafo original G_0. Sendo V_i^v o conjunto de vértices de G_i que são combinados para formar um novo vértice v único, chamado de *multinode*, do grafo G_{i+1} contraído do próximo nível, para se manter a equivalência entre o grafo G_i e G_{i+1}, o peso do *multinode* deve ser igual à soma dos pesos dos vértices em V_i^v. Também, para preservar as informações de conectividade no grafo contraído, as arestas de v devem corresponder às arestas dos vértices em V_i^v. Caso um vértice de V_i^v contenha várias arestas para o mesmo vértice u, o peso da nova aresta (u,v) deve ser igual a soma dos pesos das arestas de V_i^v para u, como mostrado na Figura 4.2. Desta forma, o corte de arestas da partição em um grafo contraído será igual ao corte de arestas da mesma partição em um grafo mais refinado.



(a) Grafo original (b) Grafo contraído

Figura 4.2: Antes e depois da contração de um grafo (KARYPIS; KUMAR, 1995)

Dentre as formas de emparelhamento mais comuns, Karypis e Kumar (1995) citam:

- **Emparelhamento aleatório**: os vértices são visitados em ordem aleatória. Se um vértice u ainda não foi emparelhado, então seleciona-se aleatoriamente um de seus vértices adjacentes não emparelhado. Se existe tal vértice v, inclui-se a aresta (u,v) no emparelhamento e marca-se os vértices u e v como emparelhados. Se não houver vértices adjacentes não emparelhados ao vértice v, então o vértice u se mantém não emparelhado.

- **Emparelhamento de arestas pesadas**: baseado na ideia que o particionamento deve encontrar uma solução que minimiza o corte de arestas, então, contrair as arestas com maior peso, resulta em um grafo que possuirá somente arestas com peso menor, permitindo que o particionamento já inicie seu processo com estas arestas. Os vértices também são visitados aleatoriamente, porém a aresta (u, v) escolhida é a que possui maior peso em relação a todas arestas incidentes a v.
- **Emparelhamento de arestas leves**: consiste em encontrar um emparelhamento onde as arestas contraídas tenham o menor peso, levando a uma menor redução nos pesos das arestas do grafo G_{i+1}. Karypis e Kumar (1995) mencionam que esta técnica aumenta a média do grau de G_{i+1} em relação à G_i, e que isto é importante para certas heurísticas de particionamento tal como Kernighan e Lin (1970), pois elas produzem boas partições em pouco tempo para grafos com altos valores médios de grau dos vértices. Sua execução emparelha o vértice u com o vértice v tal que o peso da arestas (u, v) seja mínimo.

2. **Particionamento**: consiste em executar um particionamento *2-way* de alta qualidade (isto é, pequeno corte de arestas), tal que cada parte contenha aproximadamente metade dos pesos dos vértices do grafo original. Desde que o grafo G_n contém informação suficiente para garantir os requisitos de um particionamento balanceado e um pequeno corte de arestas, ele pode ser particionado usando vários algoritmos *2-way*, visto que o grafo agora possui uma quantidade reduzida de vértices.

3. **Expansão**: este processo consiste em projetar a partição P_n de G_n de volta para G_0, indo através das partições intermediárias P_{n-1}, P_{n-2}, ..., P_0. Isso é possível pois cada vértice de G_{i+1} contém um subconjunto distinto de vértices de G_i, permitindo projetar o particionamento P_i a partir de P_{i+1}.

   Sendo P_{i+1} um particionamento mínimo local, P_i pode não necessariamente ser local, pois é mais refinado e possui um grau maior de liberdade que pode ser usado para melhorar ainda mais o particionamento através de uma heurística de refinamento local.

4.3.1 Refinamento local

O propósito básico de um algoritmo de refinamento *2-way* é selecionar um subconjunto de vértices de cada partição, tal que quando trocados resulte em uma nova partição com um corte de arestas menor. Esses subconjuntos consistem de vértices que participam do corte atual, sendo que o restante dos vértices já foram processados nos níveis anteriores e não precisam ser mais examinados.

De acordo com Karypis e Kumar (1995), o algoritmo de refinamento *Boundary Kernighan-Lin* (BKL) é um dos algoritmos baseados na heurística KL que produzem bons resultados neste processo.

Essa estratégia consiste em somente calcular os ganhos dos vértices que estão na fronteira do particionamento. Similar ao algoritmo Kernighan e Lin (1970), após a troca de um vértice v, é necessário atualizar o ganho de seus vértices adjacentes, que ainda não foram trocados (agora sim, são incluídos inclusive os vértices adjacentes a v mesmo que esses não estejam na fronteira). Se qualquer desses vértices adjacentes tornar-se fronteira devido a troca de v, ele deve ser inserido na estrutura de dados somente se ele possuir ganho positivo. O custo de execução desse refinamento é reduzido devido à quantidade de vértices envolvidos no processamento.

4.4 Greedy K-Way

Esse algoritmo *k-way*, definido por Jain, Swamy e Balaji (2007), faz o particionamento de um grafo a partir da escolha aleatória de k vértices iniciais, onde, no decorrer da execução, os vértices remanescentes são adicionados alternadamente a cada partição, de forma que em cada estágio o vértice adicionado seja aquele que resulta em um aumento mínimo no corte.

Esse processo se baseia em algumas abordagens gulosas de particionamento *2-way*, utilizadas por Jain, Swamy e Balaji (2007):

- ***Standard Greedy***: a partir de dois vértices iniciais, os outros vértices são adicionados alternativamente nas duas partições resultando em um aumento mínimo no corte.
- ***Min-Max-Greedy***: adiciona uma regra de desempate no esquema *Standard Greedy*, onde ao adicionar um vértice na partição P, o vértice escolhido

é aquele com mais vizinhos em P. A razão dessa escolha é que quanto mais arestas se tornam internas, menos arestas serão externas nas iterações seguintes.

- ***Diff-Greedy***: usa um critério de seleção de um vértice onde minimiza a diferença entre as novas arestas que cruzam o corte e as arestas internas, onde novas arestas que se tornam internas têm prioridade maior.

Essas abordagens foram usadas como ponto de partida para o desenvolvimento do algoritmo guloso de particionamento *k-way*.

Os autores definem algumas notações que serão usadas na execução do algoritmo *k-way*. Dado $i \in V$, $X(i)$ é o conjunto ao qual o vértice i pertence e

$$N(i,p) = |\{(i,j) : j \in p\}| \tag{4.9}$$

é número de arestas incidentes em i na qual a outra ponta da aresta está no conjunto p. Então define-se:

$$Ext(i,p) = \sum_{\substack{j=0 \\ j \neq p}}^{k-1} N(i,j) \tag{4.10}$$

sendo a quantidade de arestas externas ao conjunto p caso o vértice i seja adicionado a p, e

$$Diff(i,p) = Ext(i,p) - N(i,p) \tag{4.11}$$

é a diferença entre as arestas adicionais introduzidas no corte e as novas arestas internas de p.

Com essas definições, os autores estendem a ideia do *Diff-Greedy* para definir o *K-Greedy* utilizando o *MinP-Greedy* e *MaxN-Greedy*, definidos a seguir, onde, para adicionar a um conjunto p, o vértice i é escolhido tal que o $Diff(i,p)$ seja mínimo. Isso mantém a lógica que quanto mais arestas se tornam internas, menos arestas irão cruzar o corte nas próximas iterações.

- ***K-Greedy***: é uma extensão do *Diff-Greedy* para múltiplas partições. Ele consiste em colocar k vértices aleatórios em K subconjuntos, incorporando

os vértices com mais vizinhos no subconjunto que será adicionado. Ele usa *MaxN-Greedy* para selecionar o vértice.

- ***MinP-Greedy***: esta estratégia seleciona o próximo subconjunto a ser usado para adicionar um vértice. Para todo conjunto j, define $minval(j)$ sendo o valor mínimo de $Diff(i,j)$ sobre todos os vértices i remanescentes. Então o grupo escolhido é tal que $minval(p) = min_{j \in 0,1,\ldots,k-1} minval(j)$. A escolha do vértice é feita aleatoriamente entre aqueles com o valor $Diff$ igual a $minval(p)$.

- ***MaxN-Greedy(neighbour)***: adiciona uma regra de desempate − o conjunto de vértices com mínimo valor de $Diff(i, addset)$ são reduzidos para um subconjunto contendo vértices com o maior número de vizinhos em $addset$. Então, um vértice é selecionado aleatoriamente desse grupo. Essa estratégia dá um passo a mais em relação ao *K-Greedy*, pois usa os pesos internos das arestas.

Finalmente, a heurística *MinP-MaxN Greedy*, mostrada no Algoritmo 3, combina ambos *MinP-Greedy* e *MaxN-Greedy* escolhendo p, sobre o qual um vértice será adicionado, baseado no valor $minval(p)$. Tal vértice i é selecionado com base em $diff(i,p)$ e $N(i,p)$ e então ele procede gulosamente até consumir todo o grafo.

Algoritmo 3: Algoritmo *MinP-MaxN Greedy* (JAIN; SWAMY; BALAJI, 2007)

Entrada: Um conjunto de vértices V
Saída: O tamanho do corte f

```
1  restante ← V
2  f ← 0
3  v ← vértice aleatório (rv) ∈ restante
4  set(0) ← {v}
5  restante ← restante \ {v}
6  enquanto |restante| > 0 faça
7      para todo j ∈ {0,...,k−1} faça
8          minval(j) ← min_{i∈restante} Diff(i,j)
9      addset ← min_{|set(j)|<avg(subset size)} minval(j)
10     v ← rv ∈ {i : Diff(i,addset) = minval, N(i,addset) seja máximo}
11     set(addset) ← set(addset) ∪ {v}
12     restante ← restante \ {v}
13     f ← f + Ext(v,addset)
14 retorna f
```

4.5 *Greedy Iterative Improvement (Greedy IIP)*

Nesta heurística, Becker et al. (2001) demonstraram que a técnica de melhoramento iterativa gulosa é poderosa o suficiente para resultar em um eficiente algoritmo de particionamento de grafos, particularmente bem adequado para lidar com grandes quantidades de partições.

O Algoritmo 4 mostra um esboço da sua estrutura principal, no qual possui uma característica multinível devido ao fato de juntar os vértices antes de iniciar o particionamento.

Algoritmo 4: Execução principal *Greedy IIP* (BECKER et al., 2001)

```
Entrada: O grafo, o número de partições e o parâmetro runs
Fazer o pré-processamento do grafo
Criar uma partição inicial P' usando um algoritmo BFS
Setar mingain < 0
enquanto mingain ≤ 0 faça
    r ← runs
    repita
        P ← P'
        P' ← refinar(P, k, mingain)
        se custo(P) ≤ custo(P') então
            r ← r − 1
    até r < 0
    incrementar mingain
Executar a fase de expansão e refinamento
```

O fluxo geral do algoritmo pode ser descrito em quatro componentes:

1. **Pré-processamento e geração da partição inicial**: consiste no emparelhamento dos vértices adjacentes com grau menor que três e usa *Breath First Search* (BFS) para definir a partição inicial.

2. **Refinamento guloso (*core*)**: move um vértice após o outro (se o ganho for maior que o valor de *mingain*) para a *melhor* partição vizinha (obtida através do máximo ganho). O parâmetro *mingain* permite uma escalada restrita para passos locais.

3. **Melhoramento Iterativo**: conforme mostrado no Algoritmo 5, ele inicializa *mingain* com o valor negativo da média do grau dos vértices do grafo e iterativamente move os vértices permitidos pelo valor do *mingain* estipulado.

Algoritmo 5: Refinamento do *Greedy IIP* (BECKER et al., 2001)

Entrada: A partição inicial P, o número de partições k e o parâmetro *runs*
Saída: A partição P

```
1  para cada vértice n faça
2      fazer C_i ser a partição a qual o vértice n pertence
3      se |C_i \ {n}| > L então
           // L é o limite inferior do tamanho da partição
4          para todo conjunto C_j (j ≠ i) faça
5              se |C_j ∪ {n}| ≤ R então
                   // R é o limite superior do tamanho da partição
6                  seta P' para a partição depois de mover n de C_i para C_j
7                  seta ganho(C_j) para (custo(P) − custo(P'))
8              senão// descarta o vértice n para a partição C_j
9                  seta ganho(C_j) para −∞
10         se max(gain(C_j)) = gain(C_l) ≥ mingain então
11             mover n de C_i para C_l
12 retorna P
```

A cada execução o valor de *mingain* é incrementado para reduzir a flexibilidade de movimentos dos vértices. O algoritmo utiliza o parâmetro *runs* para limitar o número de passos sem que haja melhorias no refinamento para um valor fixo de *mingain*.

4. **Pós-processamento**: executa a expansão dos vértices fazendo uma sequência final de passos de refinamento, sendo que, no último passo da fase de melhoramento iterativo, não é permitida a redução de qualidade do particionamento.

4.6 *Fast unfolding of communities*

Este método, proposto por Blondel et al. (2008), encontra partições de alta modularidade, que se desdobra em uma hierarquia completa de grandes grafos, possibilitando diferentes resoluções de detecção de comunidades. O algoritmo é baseado no ganho de modularidade ΔQ que é obtido movendo-se um vértice i para uma comunidade C, sendo definido por:

$$\Delta Q = \left[\frac{\sum_{in}+k_{i,in}}{2m} - \left(\frac{\sum_{tot}+k_i}{2m}\right)^2\right] - \left[\frac{\sum_{in}}{2m} - \left(\frac{\sum_{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right] \qquad (4.12)$$

onde $\sum_{in}$ é a soma dos pesos das arestas dentro de C, $\sum_{tot}$ é a soma dos pesos das arestas incidentes aos vértices em C, k_i é a soma dos pesos das arestas incidentes ao vértice i, $k_{i,in}$ é a soma dos pesos das arestas de i para vértices em C, m é a soma dos pesos de todas as arestas do grafo.

O processo, descrito no Algoritmo 6, é dividido em duas fases que são repetidas iterativamente. A primeira fase agrupa os vértices em comunidades, enquanto que a segunda fase constrói um novo grafo no qual cada novo vértice corresponde a uma das comunidades encontradas na primeira fase, sendo que o peso de cada vértice resultante é a soma dos pesos dos vértices da comunidade correspondente.

Algoritmo 6: Algoritmo *Fast Unfolding* (BLONDEL et al., 2008)

```
   Entrada: Um grafo G
   Saída: A partição P
1  enquanto houver melhorias na modularidade faça
       // Primeira fase
2      Atribuir diferentes comunidades para cada vértice
3      para cada vértice i faça
4          para cada vizinho j de i faça
5              Avaliar o ganho de modularidade removendo i de sua
               comunidade e colocando ele na comunidade de j
6          Colocar o vértice i na comunidade para a qual o ganho(positivo) é
           máximo
       // Segunda fase
7      Construir um novo grafo, onde, cada vértice corresponde às
       comunidades encontradas na primeira fase
8  retorna P
```

O processo composto pelas duas fases anteriores é chamado de *passo*, pois após a contração feita pela segunda fase, basta executar a primeira fase utilizando o novo grafo.

A cada *passo*, o número de meta-comunidades (partições) diminui e são executados até não haver mais mudanças e o máximo de modularidade for alcançado.

Entre as vantagens citadas por Blondel et al. (2008), aparecem a facilidade e intuição de implementação, a rapidez de execução, pois possui uma natureza intrínseca multinível, apesar que, segundo Fortunato e Barthelemy (2006), a otimização da modularidade falha em identificar comunidades menores que um certo tamanho.

4.7 *Multilevel Banded Diffusion*

O esquema de difusão proposto por Pellegrini (2007) apresenta um modo de integrar a otimização e a difusão global em uma abordagem multinível de *banda* para fazer um particionamento *2-way* representando um grafo, como mostrado na Figura 4.3.

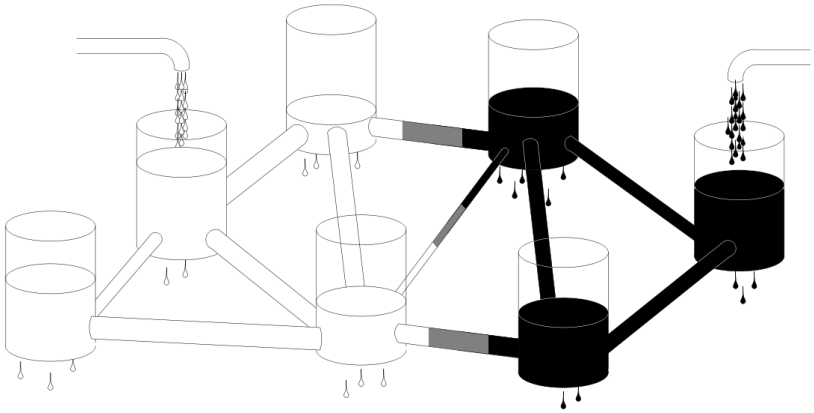


Figura 4.3: *The jug of the Danaides* (PELLEGRINI, 2007)

No algoritmo, chamado pelo autor de *The jug of the Danaides*, os vértices são barris de capacidade infinita, que vaza tal que, no máximo, uma quantidade de líquido goteja por unidade de tempo. Quando os vértices possuem peso, a quantidade de líquido perdida por unidade de tempo é igual ao peso do vértice. As arestas são modeladas como canos de vazão igual a seu peso.

Em ambas partições, um vértice fonte é escolhido, nos quais um cano fonte é conectado, que flui em $|V|/2$ unidades de líquido por tempo, onde dois tipos de líquidos são injetados no sistema: *schoth* e *anti-schoth*. Quando os dois líquidos se misturam, eles se anulam.

Esse processo pode ser iterado até convergir ou estabilizar formando uma fronteira suave, ao invés da minimização do corte. O Algoritmo 7 mostra o detalhamento do processo de difusão dos líquidos através dos vértices do grafo.

O algoritmo também usa um esquema de refinamento multinível usando *band graph* evitando que o refinamento seja feito em todo o grafo, mas somente para uma faixa que contém vértices que estão no máximo a uma pequena distância, tipicamente três, a partir do corte projetado. O restante dos vértices de cada lado são contraídos em dois vértices *âncora*, contendo a soma dos pesos dos vértices contidos neles, assim usando esses vértices âncora como novas sementes para o es-

Algoritmo 7: Difusão *jug of the Danaides* (PELLEGRINI, 2007)

Entrada: Grafo G, Número de passos
Saída: O conteúdo dos vértices do grafo G

enquanto *número de passos a fazer* **faça**
  zerar o conteúdo do vetor *new*
  `// Recarrega os vértices fontes`
  $old[s0] \leftarrow old[s0] - |V|/2$
  $old[s1] \leftarrow old[s1] + |V|/2$
  para todo *vértice v do grafo G* **faça**
    `// obtém o conteúdo do vértice`
    $c \leftarrow old[v]$
    se $|c| > weight[v]$ **então**
      `// Se ainda há conteúdo para difundir`
      $c \leftarrow c - weight[v] * sign(c)$
      $\sigma \leftarrow \sum_{e=(v,v')} weight[e]$
      para todo *aresta* $e = (v, v')$ **faça**
        `// Distribui para os vértices adjacentes`
        $f \leftarrow c * weight[e]/\sigma$
        $new[v'] \leftarrow new[v'] + f$
  troca o conteúdo dos vetores *old* e *new*
retorna *old*

quema de difusão. Essa estratégia previne os algoritmos locais de procurar vértices longe da solução que já está próxima, processando somente os vértices perto da fronteira, ao invés do grafo todo.

4.8 Orca

A heurística *Orca Reduction and ContrAction Graph Clustering*(Orca), apresentada por Delling et al. (2009), executa localmente e hierarquicamente contraindo o grafo de entrada, evitando utilizar qualquer decisão baseada em índices, por exemplo a *modularidade*, que se revelaram ser *NP-difícil*. O Orca foi desenhado para confiar simplesmente na observação estrutural na qual traduz imediatamente a densidade *intra-cluster* e a esparcialidade *inter-cluster*. O Orca se parece com o algoritmo proposto por Blondel et al. (2008), mas sem direcionar-se à modularidade.

Delling et al. (2009) apresentam alguns argumentos justificando o uso de métodos locais de particionamento, citando alguns fatos que encorajam o uso destes métodos:

- mostram um comportamento escalável;
- operações locais no grafo permitem estruturas de dados mais rápidas;
- estratégias locais são mais adequadas para dinamização.

A abordagem geral do algoritmo Orca é a seguinte: desbastar os vértices irrelevantes do grafo, então iterativamente identificar e contrair as vizinhanças densas em *super-nodes*, repetindo a contração no próximo nível de hierarquia. Se houver alguma falha na contração, remove-se os vértices de baixo grau e troca-os por arestas, chamadas de *atalhos*. O Algoritmo 8 mostra a estrutura principal do Orca.

Algoritmo 8: Abordagem geral do algoritmo Orca (DELLING et al., 2009)

```
   Entrada: Grafo G(E,V,w), d, γ ∈ ℝ+
1  Core-2 Reduction(G)
2  enquanto |V| > 2 faça
3  |  Dense-Region-Global(G, γ, d)
4  |  se |V_old| > 0.25|V| então
5  |  |  ShortCuts(G, δ)
6  |  senão
7  |  |  StoreCurrentClustering
```

O algoritmo utiliza o conceito de *vizinhança* de um vértice que é dada por:

$$N(v) = \{w \in V | \{v, w\} \in E\} \tag{4.13}$$

juntamente com a definição de $d-$*neighborhood*, que corresponde ao conjunto de vértices vizinhos de v a uma distância máxima d, e é definida por:

$$N_d(v) = \{w \in V | w \neq v, dist(v, w) \leq d\} \tag{4.14}$$

onde $dist(v, w)$ é o comprimento do menor caminho entre v e w.

O detalhamento de cada função chamada nesses passos são definidas a seguir:

1. **Redução *Core-2***: o *core-2* de um grafo, definido por Seidman (1983), é um subgrafo induzido máximo, com vértices com no mínimo grau 2. Sua base lógica é: vértices com somente uma aresta são apêndices como folhas em árvores. Então, como esses apêndices não são muito grandes em grafos

reais, os autores assumem que estes apêndices sejam anexados junto com seus vértices âncora.

2. **Busca local de regiões densas**: região densa $R \subseteq V$ é um conjunto de c vértices com distância d de algum vértice inicial v tal que cada vértice $w \in R$ está no máximo a uma distância d de pelo menos $|N_d(v)|/\gamma$ outros vértices de $N_d(v)$. Esta lógica de busca descrita no Algoritmo 9 utiliza o atributo *seen* para armazenar quantos vértices de $N_d(w)$ são considerados $d-neighbor$, onde valores baixos de γ impõem um critério restrito de densidade, o qual leva a função *Dense-Region-Local* retornar regiões pequenas.

Algoritmo 9: Algoritmo *Dense Region Local* (DELLING et al., 2009)

Entrada: Grafo G, vértice inicial v, grau de densidade γ, tamanho da vizinhança a ser explorada d

```
1 Inicializa a região com v
2 para cada vértice w a uma distância d ou menos de v faça
3 |   para cada vértice x ∈ N_d(w) faça
4 |   |   incrementar o atributo seen de x
5 Adiciona cada vértice w ∈ N_d(v) na região, os quais foram vistos por pelo
  menos a uma fração γ dos vértices em N_d(v)
```

3. **Contração de regiões densas**: faz a contração do subgrafo encontrado no item 2 anterior em um *super-node*.

4. **Detecção global de regiões densas**: orquestra as chamadas para as duas operações locais 2 e 3, reduzindo rapidamente o tamanho do grafo. Ele chama o Algoritmo 9 para cada vértice x, armazenando cada região densa em uma fila de prioridades com uma chave que expressa quão significante essa região é. Essa chave é calculada através da média dos pesos das arestas incidentes em um vértice na região, dada por:

$$\psi = \frac{\sum_{e \in E(D)} w(e)}{|D|} \tag{4.15}$$

onde $D \in V$ são os vértices de uma certa região.

Após determinar e enfileirar a região densa para cada vértice $v \in V$, as regiões são retiradas da fila e contraídas.

5. **Densificação via atalhos**: consiste em trocar um vértice de baixo grau δ, onde todos os pares $v1, v2$ de vértices adjacentes a V tornam-se conectados

e então V é removido. Segundo os autores esse passo é pouco usado.

Os valores dos parâmetros utilizados no algoritmo são definidos pelos autores como: $d = 1$, $\delta \leq 2$ e $2 \leq \gamma \leq 10$, sendo que baixos valores de γ são melhores no geral.

4.9 Espectral *K-Cut*

A proposta feita por Ruan e Zhang (2007) é um algoritmo espectral híbrido dos métodos *k-way* direto e *2-way* recursivo, para minimizar o corte através do cálculo dos autovalores e autovetores da matriz laplaciana do grafo.

O algoritmo **K-Cut** faz uso da estratégia recursiva do algoritmo proposto por Shi e Malik (1997) para fazer a bipartição do grafo (para mais de duas partições, executa-o recursivamente) usando a conectividade algébrica da matriz laplaciana L do grafo, seguindo os seguintes passos:

1. computar o autovetor $\mu 2$, o segundo menor autovetor generalizado de L;
2. conduzir uma busca linear em $\mu 2$ para encontrar a partição do grafo;
3. aplicar recursivamente os passos 1 e 2, caso seja necessário mais de duas partições.

Outro algoritmo espectral utilizado no **K-Cut** é o NJW (NG; JORDAN; WEISS, 2002), que procura um particionamento *k-way* de um grafo diretamente, onde k é um parâmetro informado pelo usuário. Ele utiliza o algoritmo *k-means* (ELKAN, 2003) e seu funcionamento segue os seguintes passos:

1. computar os k autovetores generalizados com menores autovalores μ de L e empilhá-los em colunas para formar a matriz $Y = [\mu 1, \mu 2, \ldots, \mu k]$;
2. normalizar cada linha de Y para ter comprimento unitário;
3. tratar cada linha como um ponto em R^k e aplicar o algoritmo *k-means* para agrupá-los em k *clusters*.

Os autores usam a modularidade definida por Newman e Girvan (2003) como

$$Q(\Gamma^k) = \sum_{i=1}^{k} \left(\frac{e_{ii}}{c - (a_i/c)^2} \right) \tag{4.16}$$

onde Γ^k é uma partição do grafo, k é o número de partições, Q é o valor da modularidade, e_{ii} é o número de arestas com ambos vértices na comunidade i, a_i é o número de arestas com um ou mais vértices na comunidade i e c é o número total de arestas. Valores altos de Q indicam que a partição possui uma estrutura de comunidade forte.

Um outro algoritmo espectral utilizado no **K-Cut** é o WS (WHITE; SMYTH, 2005), que a partir de um valor k, procura o melhor valor de Q.

Para determinar automaticamente o número de comunidades, este algoritmo é executado várias vezes, com k variando de um valor K_{min} até um número máximo K_{max} de comunidades. O valor de k que fornecer o maior valor de Q é considerado o mais apropriado número de comunidades. A seguir é mostrado os dois principais passos da execução do algoritmo WS:

1. Para cada k, $k_{min} \leq k \leq k_{max}$, aplicar NJW para encontrar um particionamento *k-way*, denotado por Γ_k

2. Fazer $k^* = argmax_k\ Q(\Gamma_k)$ ser o número de comunidades e $\Gamma^* = \Gamma_{k^*}$ ser a melhor estrutura de comunidade, ou seja, o melhor particionamento.

Apesar da facilidade do WS encontrar boas estruturas de comunidades, ele é mal dimensionado para grafos grandes, pois o algoritmo *k-means* necessita ser executado até k_{max} vezes, sendo que ele se torna impraticável ao iterar para todos os $k's$ possíveis em grafos grandes e densos.

O algoritmo *K-Cut* executa os passos mostrados no Algoritmo 10.

Ele usa a combinação das duas abordagens (o particionamento recursivo e com os métodos diretos *k-way*), iniciando com o cálculo do melhor particionamento *k-way* com $k = 2, 3, \ldots, l$, restringindo l a um valor baixo (3 ou 4), usando o algoritmo NJW e selecionando o k que forneça o maior valor de Q. Então, para cada partição, o algoritmo é recursivamente aplicado.

Algoritmo 10: Algoritmo KCut (RUAN; ZHANG, 2007)

Entrada: grafo G, parâmetro l

Inicializar Γ para ser um *cluster* unitário com todos os vértices V
Setar $Q = 0$
para cada *cluster* P *em* Γ **faça**
    Fazer g ser a subrede de G contendo os vértices em P
    para cada *inteiro* k *de 2 até* l **faça** `// iteração WS`
        Aplicar NJW para encontrar um particionamento k-*way* de g: Γ_k^g
        Computar o novo valor de Q da rede como $Q'_k = Q(\Gamma \cup \Gamma_k^g \setminus P)$
    Encontrar o k que forneça o melhor valor de Q, isto é, $k^* = argmax_k\ Q'_k$
    se $Q'_{k^*} > Q$ **então**
        Aceitar o particionamento substituindo P com $\Gamma_{k^*}^g$, isto é, $\Gamma = \Gamma \cup \Gamma_{k^*}^g \setminus P$
        Atribuir $Q = Q'_{k^*}$
    Avançar para o próximo *cluster* em Γ, se existir mais algum

4.10 DiDiC

Processos difusivos podem ser usados para modelar importantes fenômenos de transporte tais como fluxo de calor, movimento de partículas e propagação de doenças.

Nesta direção, o algoritmo, proposto por Gehweiler e Meyerhenke (2010), chamado de *Distributed Diffusion Clustering* (DiDiC), utiliza do esquema de difusão para fazer o particionamento do grafo e é motivado pela sua proximidade com *caminhos aleatórios* (LOVÁSZ, 1993), pois em ambos os processos é provável que se fique um bom tempo em uma região densa do grafo, antes de sair dessa região.

De acordo com os autores, o processo difusivo para atribuir vértices às partições ocorre da seguinte forma: associa-se k valores de carga de difusão para cada vértice, sendo uma carga por partição. Essas cargas são distinguidas por *cores* de 1 a k. Em cada iteração o algoritmo difusivo é executado para cada *cluster* individualmente, com um tipo único de carga para cada sistema de difusão pertencente a uma partição. Em seguida, cada vértice v é atribuído para a partição para a qual v possui o maior valor de carga.

Para obter partições significantes, o método difusivo deve resultar em uma distribuição de carga desbalanceada preferencialmente com picos não muito longe dos centros anteriores das partições. Tal distribuição pode ser obtida com a introdução de uma perturbação, levando ao conceito de difusão perturbada.

Os vetores de carga, w e l (dreno) para cada vértice v, com tamanho k (número

de partições), no sistema de difusão (*cluster*) c e no tempo t é dado por $w_v^{(t)}(c)$, sendo que o vetor l, pertencente ao sistema de difusão secundário, é análogo a w.

No algoritmo, os autores inicialmente atribuem aleatoriamente os vértices às suas respectivas partições e inicializam os vetores de carga, tal que:

$$w_v^{(0)}(c) = \begin{cases} 100, & \text{para os vértices de seu próprio } \textit{cluster} \\ 0, & \text{para o restante dos vértices.} \end{cases}$$

e essa mesma inicialização é feita para o vetor l.

O conceito de *dreno*, herdado do esquema de difusão *Firt Order Scheme / Constant drain* (FOS/C) (MEYERHENKE; MONIEN; SCHAMBERGER, 2006), é usado para perturbar o processo e desbalancear a distribuição de carga que é significativa para o particionamento. Em cada iteração, uma pequena quantia de carga (dreno) é subtraída de todos os vértices e reinserida no sistema, adicionando o dreno total igualmente dividido em um conjunto de vértices fontes $S \in V$.

O algoritmo utiliza dois esquemas de difusão:

- *Firt Order Scheme / Truncated* (FOS/T) : que calcula a carga em cada iteração $0 < s \leq \psi$ de acordo com:

$$x_{e=\{u,v\}}^{(s-1)}(c) = \omega(e) \cdot \alpha(e) w_u^{(s-1)}(c) - w_v^{(s-1)}(c) \tag{4.17}$$

  e

$$w_u^{(s)}(c) = w_u^{(s-1)}(c) - l_u^{(s)}(c) + \sum_{e=(u,v)\in E} x_e^{(s-1)}(c) \tag{4.18}$$

  onde c indica o sistema de difusão, s indica a iteração FOS/T, x_e é o fluxo de carga via aresta e, w_u é a carga do vértice u, $\omega(e)$ é o peso da aresta e, $\alpha(e)$ é a escala de fluxo da aresta e, l_u é o vetor de carga do sistema de difusão secundário do vértice u.

  Para $\alpha(e)$, os autores propuseram o valor de $1/max\{deg(u), deg(v)\}$,evitando que uma grande carga seja levada de um lado para o outro indefinidamente.

  Usando os pesos dos vértices, chamado de *benefits*, o sistema secundário direciona a carga do sistema i rapidamente para os vértices da $i-$ésima partição.

- *Firt Order Scheme / Benefits* (FOS/B) : usa os benefícios do vértice para distribuir a carga, executando em cada iteração $0 < r \leq \rho$

$$y_{e=\{u,v\}}^{(r-1)}(c) = \omega(e) \cdot \alpha(e) \left(\frac{l_u^{(r-1)(c)}}{b_u(c)} - \frac{l_v^{(r-1)}(c)}{b_v(c)} \right) \tag{4.19}$$

e

$$w_u^{(s)}(c) = w_u^{(s-1)}(c) - l_u^{(s)}(c) + \sum_{e=(u,v)\in E} y_e^{(s-1)}(c) \tag{4.20}$$

onde c indica o sistema de difusão, r indica a iteração FOS/B, y é o fluxo de carga via aresta e, w_u é a carga do vértice u, $\omega(e)$ é o peso da aresta e, $\alpha(e)$ é a escala de fluxo da aresta e (sutilmente escolhida pelos autores), l_u é o vetor de carga do sistema de difusão secundário e $b_u(c)$ é o benefício do vértice u na partição π_c tal que

$$b_u(c) = \begin{cases} 1, & u \notin \pi_c \\ B \gg 1, & \text{caso contrário} \end{cases}$$

Os valores de benefício $b_u(c)$ e $b_v(c)$ (com valor 10 baseados nos experimentos) assegura que vértices que não estão no *cluster* i enviam a maioria de sua carga de l para seus vizinhos na partição i.

O Algoritmo 11 mostra os passos, executando T vezes o *time step*, que por sua vez executa ρ vezes o esquema FOS/B dentro de um *loop* FOS/T.

Algoritmo 11: Algoritmo DiDiC (GEHWEILER; MEYERHENKE, 2010)

Entrada: vértice V, vizinhança $N(V)$, π, k, T, ψ, ρ

```
se π é indefinido então
    π ← randomValue(1,k)
w ← setarCargaInicial(π)
para t até T faça
    para cada sistema de partição c faça
        para s = 1 até ψ faça // iteração FOS/T
            para r = 1 até ρ faça // iteração FOS/B
                para cada vizinho u em N(v) faça
                    l_v(c) = l_v(c) − α(e)·ω(e)·(l_v(c)/b_v(c) − l_u(c)/b_u(c))
                para cada vizinho u em N(v) faça
                    w_v(c) = w_v(c) − α(e)·ω(e)·(w_v(c) − w_u(c))
                w_v(c) = w_v(c) + l_v(c)
    π ← argmax_{c=1,...,k} w_v(c)
    // Utilizado em grafos dinâmicos
    adaptarMudançasGrafo(v, N(v)) → N(v)
```

No fim de cada *time step*, o algoritmo permite que os valores de carga sejam ajustados caso o grafo tenha sido alterado, permitindo que grafos dinâmicos sejam utilizados nesse processo. Grafos dinâmicos não serão discutidos aqui, pois estão fora do escopo desse trabalho.

4.11 Considerações finais

Além dos algoritmos apresentados neste capítulo, outras heurísticas desenvolvidas para encontrar os resultados de uma solução aceitável podem ser vistas em Osipov e Sanders (2010), Meyerhenke e Sauerwald (2012) e Delling et al. (2011).

Mais informações sobre algoritmos de particionamento de grafos podem ser vistas em Fortunato (2010) e Schaeffer (2007).

Uma parte da dificuldade encontrada no trabalho foi definir uma estrutura de dados do grafo para efetuar a implementação e os testes dos algoritmos levando em consideração os objetivos do trabalho, que é dar independência ao código em relação à forma de armazenamento do grafo. Isto levou à necessidade da definição de uma estrutura de classes organizada em uma arquitetura de *software* para abstrair o acesso ao grafo, dando origem ao conteúdo apresentado no próximo capítulo.

5 Arquitetura

Os capítulos anteriores proveram um embasamento teórico sobre banco de dados orientados a grafos e algoritmos de particionamento.

Este capítulo aborda o principal objetivo do trabalho que é a proposta de uma arquitetura de *software*, que 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 importante estruturar os principais componentes em uma solução de *software* através de seus módulos. Por questão de abrangência desta solução, todos os termos, nomes e expressões utilizados nos diagramas foram definidos na língua inglesa.

5.1 Visão geral da solução

A fim de facilitar o desenvolvimento e a manutenção, bem como aumentar a modularidade, a arquitetura foi projetada em três módulos distintos, cada qual com sua responsabilidade, provendo um ambiente que favorece a criação de algoritmos de particionamento de grafos, conforme ilustra a Figura 5.1.

O componente *algorithm* é responsável pela lógica de execução do algoritmo desejado, bem como manter as estruturas de dados pertencentes à sua execução interna. O componente *partition* 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. Este componente provê suporte para o componente *algorithm* de forma que este se preocupe somente com a lógica necessária para definir em qual partição um vértice deve ser adicionado ou removido.

O componente *graph* possui as classes e interfaces genéricas que representam

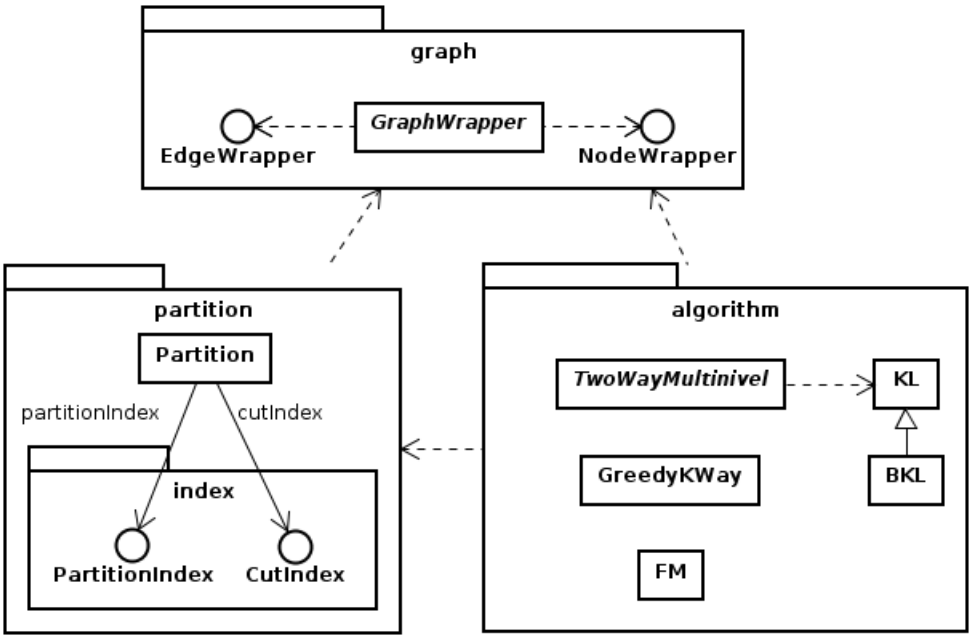


Figura 5.1: Visão geral da arquitetura proposta

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

Já no componente *partition* se encontram as estruturas e operações de atualização e cálculo do valor do corte, além de classes de indexação de vértices e arestas, contidas no subcomponente *index*, que facilita a obtenção do conjunto atual de vértices de uma dada partição, a inserção e a remoção de vértices de seus respectivos índices e a obtenção das arestas do corte atual e o valor do corte de arestas.

Com este modelo de componentes, a arquitetura 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.

Através da descrição das classes do modelo desenvolvido, os principais elementos específicos e genéricos da arquitetura, e seus relacionamentos, serão apre-

sentados 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. A arquitetura 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, a arquitetura pode ser estendida para trabalhar com outros bancos de dados orientados a grafos.

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

5.2 Generalização da estrutura do grafo

Para que um algoritmo seja executado utilizando um grafo de forma genérica, foi definida uma estrutura padrão, ilustrada na Figura 5.2, independentemente de usar objetos provenientes da estrutura em memória ou do banco.

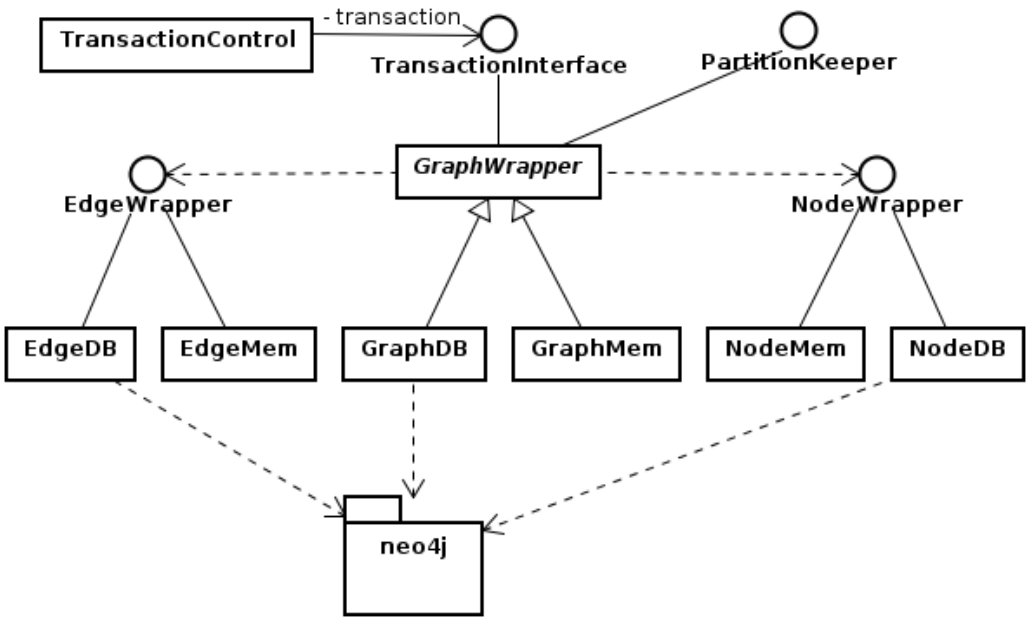


Figura 5.2: Generalização da estrutura do grafo

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 da arquitetura. É a partir desta estrutura que o algoritmo de particionamento deve fazer o acesso aos elementos do grafo.

A classe central deste componente é `GraphWrapper` que é definida de forma genérica utilizando as interfaces `NodeWrapper` e `EdgeWrapper`. Ela realiza a interface `TransactionInterface` para permitir a padronização do controle de transações, que é utilizada pela classe `TransactionControl` e também realiza a interface `PartitionKeeper` para manuseio dos índices internos de particionamento.

A classe `GraphWrapper`, detalhada na Figura 5.3, 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(...)`.

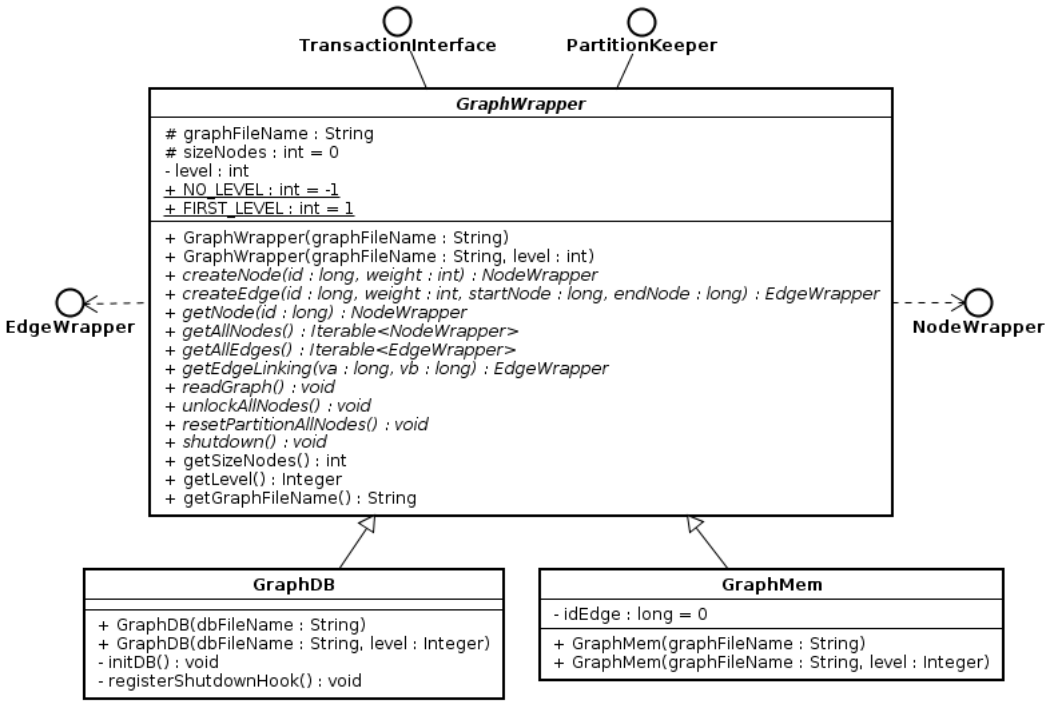


Figura 5.3: Detalhes de `GraphWrapper`, `GraphDB` e `GraphMem`

Também é possível obter um vértice através de seu identificador, utilizando o método `getNode(long id)` ou obter todos os vértices e arestas, através dos métodos `getAllNodes()` e `getAllEdges()`, respectivamente. Estes métodos são abstratos para permitir que os detalhes de armazenamento, leitura e manipulação sejam feitos por suas subclasses, `GraphDB` e `GraphMem`.

Além destes métodos básicos de manuseio de vértices e arestas, ainda foram

definidos os seguintes métodos:

- `getEdgeLinking(...)`: obtém a aresta que conecta dois vértices.
- `readGraph()`: faz a inicialização das estruturas internas do grafo.
- `unlockAllNodes()`: marca o atributo `lock` de todos os vértices com o valor `false`.
- `resetPartitionAllNodes()`: anula o valor da partição de cada vértice.
- `shutdown()`: permite ao algoritmo fazer o encerramento da fonte de dados do grafo de forma transparente, porém somente é implementado para fazer as rotinas internas de finalização do banco de dados Neo4J.

O parâmetro `graphFileName` dos construtores, que especifica o arquivo ou o diretório do grafo original, dependendo de sua subclasse, é usado pelo método `readGraph()` e o parâmetro `level` é utilizado por algoritmos multiníveis. Os atributos são acessados via métodos de acesso (`get`) do padrão Java, onde a quantidade de vértices existentes no grafo é obtida com o método `getSizeNodes()`. Cada subclasse implementa sua própria forma de armazenamento dos valores de propriedades correspondente ao objeto desejado. Os métodos implementados por `GraphDB` e `GraphMem`, herdados de sua superclasse e interfaces, foram omitidos por questões de facilidade de leitura do diagrama.

Conforme ilustrado na Figura 5.4, a classe `GraphDB` implementa os métodos definidos pela classe abstrata `GraphWrapper` de forma que toda a operação feita no grafo seja efetivada no banco de dados Neo4J, acessado através do atributo `graphFileName`, por uma instância da interface `GraphDatabaseService`. Esta por sua vez possui métodos para criar e remover vértices e arestas do banco, recuperar todos os vértices ou todas as arestas existentes, manipular índices criados a partir de vértices ou arestas, através das interfaces `RelationshipIndex` e `Index` e manipular as transações do banco de dados, através da interface `Transaction`. O método `initDB()` faz a inicialização interna do banco de dados, realizando uma chamada ao método `registerShutdownHook()` que permite à execução finalizar o banco caso ocorra uma terminação inesperada na execução no algoritmo.

Ainda de acordo com a Figura 5.4, as classes `GraphDB`, `NodeDB` e `EdgeDB` utilizam as interfaces `GraphDatabaseService`, `Node`, `Index`, `Relationship`,

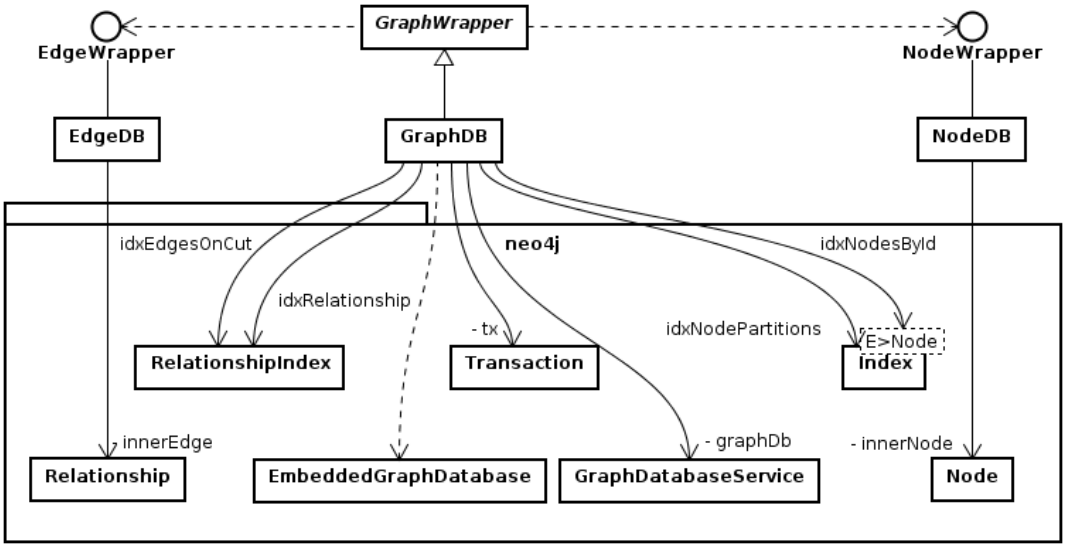


Figura 5.4: Uso das classes do Neo4J

`RelationshipIndex`, `Transaction`, e a classe `EmbeddedGraphDatabase` do pacote `neo4j` que são implementadas pelo Neo4j. Já as classes `GraphMem`, `NodeMem` e `EdgeMem` armazenam o grafo em seus atributos, mantendo em memória os objetos que compõem a estrutura do grafo, que é lido a partir do arquivo especificado pelo atributo `graphFileName`.

Para auxiliar na manipulação do grafo, foram definidas duas interfaces e uma classe, ilustradas na Figura 5.5, para suporte à transação e criação dos índices internos.

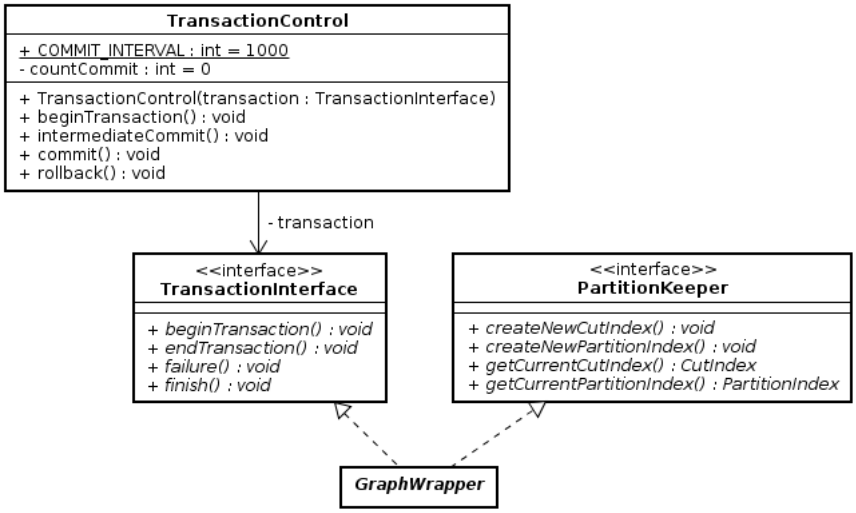


Figura 5.5: Suporte para transações e índices

A interface `TransactionInterface`, que permite o controle de transações, define os serviços de início e fim de uma transação, bem como a ocorrên-

cia de falha ou o sucesso da mesma. Estes métodos, herdados através da classe `GraphWrapper`, são implementados pelas classes `GraphDB` e `GraphMem`, onde a classe `GraphDB` delega as chamadas a uma instância de `Transaction` do banco Neo4J e a classe `GraphMem` não trata situações transacionais, pois qualquer problema na execução, implica no carregamento do grafo original novamente.

Devido ao volume de informações esperado para ser processado pelos algoritmos, foi definida também a classe `TransactionControl`. Seu principal objetivo é efetivar as alterações intermediárias caso a quantidade de operações atingir um valor pré-definido pelo atributo `COMMIT_INTERVAL`, através do método `intermediateCommit()`. Ela também permite iniciar a transação através do método `beginTransaction()` e finalizar a transação, com sucesso ou falha, usando os métodos `commit()` ou `rollback()`, respectivamente. As chamadas destes métodos são delegadas para os métodos do atributo `transaction`, do tipo `TransactionInterface` que, pela estrutura definida, é um objeto grafo.

O uso do método `intermediateCommit()` é, normalmente, feito 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.

A interface `PartitionKeeper` define os serviços para criar e obter as instâncias das classes que realizam as interfaces `CutIndex` e `PartitionIndex`, detalhadas na seção 5.3. Para criar suas respecitvas instâncias, são definidos os métodos `createNewPartitionIndex()` e `createNewCutIndex()`. O próprio grafo fornece estes objetos através dos métodos `getCurrentCutIndex()` e `getCurrentPartitionIndex()`.

A Figura 5.6 expõe os detalhes das interfaces `NodeWrapper` e `EdgeWrapper`, que são utilizadas pela classe `GraphWrapper` para manipular o grafo, permitindo que os objetos de vértices e arestas sejam utilizados de forma genérica.

A interface `EdgeWrapper` define os métodos pertinentes ao comportamento de uma aresta, dentre os quais destaca-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, `getStartNode()` e `getEndNode()` que retornam o vértice de cada uma das extremidades da aresta e, finalmente, `getId()` que retorna o identificador da aresta. Como classes concretas, foram definidas duas subclasses,

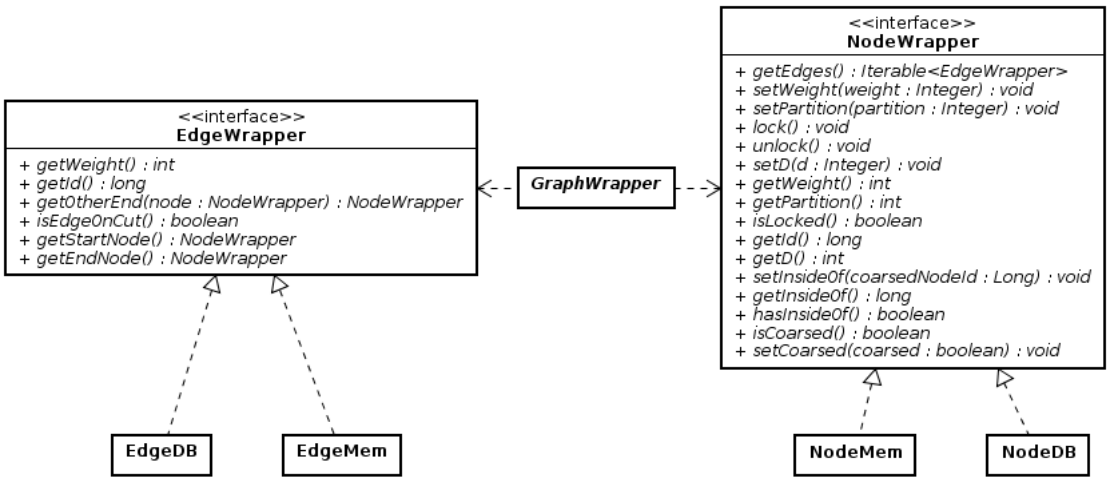


Figura 5.6: Detalhes de `NodeWrapper` e `EdgeWrapper`

`EdgeDB` e `EdgeMem` para implementar as operações de uma aresta no banco de dados e em memória, respectivamente.

A classe `EdgeDB`, que mapeia a aresta do banco Neo4J, possui o atributo `innerEdge`, do tipo `Relationship` do pacote Neo4J, para o qual delega as chamadas de todos 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 `Node` do Neo4J, no atributo `innerNode`, para o qual delega todos os métodos definidos em `NodeWrapper`. Já a classe `NodeMem` mantém as informações do vértice em seus atributos. Dentre os métodos definidos pela interface `NodeWrapper`, destacam-se os métodos `get` e `set` do padrão Java para alguns atributos já utilizados na implementação atual, como: *weight*, *D*, *partition* e *insideOf*. Ainda outros métodos foram definidos para auxiliar na execução dos algoritmos, como `getEdges()`, que obtém as arestas conectadas ao vértice em questão, `lock()`, `unlock()` e `isLocked()` para dar suporte aos algoritmos sobre quais vértices já foram utilizados em um certo momento e o `getId()` que retorna o identificador do vértice.

Para dar suporte aos algoritmos multiníveis, a interface `NodeWrapper` fornece os métodos `setInsideOf(...)`, `getInsideOf()` e `hasInsideOf()` para facilitar a contração e expansão de um grafo, onde um vértice consegue saber que

ele está contido dentro de outro vértice em um grafo mais contraído. Outros dois métodos são `isCoarsed()` e `setCoarsed(...)` para indicar a contração de um vértice, utilizado no processo de expansão do grafo.

A partir das definições de `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 da arquitetura adaptar as estruturas desses elementos para os dados específicos que ele deseja manipular durante a execução de sua lógica. Finalmente, o mesmo algoritmo pode utilizar quaisquer das duas opções fornecidas por esta arquitetura, bem como utilizar outras soluções derivadas deste.

5.3 Estrutura de particionamento

Outra característica da arquitetura é oferecer recursos para manipular as informações inerentes ao particionamento, utilizando classes e interfaces que auxiliam a manutenção dessas estruturas, construídas no pacote *partition*, cujos principais elementos são mostrados na Figura 5.7.

O principal elemento do diagrama é a classe `Partition`, que é responsável por manter os atributos *cutIndex* e *partitionIndex*, objetos do tipo `CutIndex` e `PartitionIndex` respectivamente, com as informações internas sobre um dado particionamento. Seus construtores permitem que sejam definidos o número k de partições, armazenado internamente e acessado pelo método `getK()`, aumentando a flexibilidade da arquitetura, propiciando seu uso por algoritmos *k-way* e os dois parâmetros *cutIndex* e *partitionIndex* que são armazenados em seus respectivos atributos. O segundo construtor, com o parâmetro `level`, torna possível o uso desta classe por algoritmos multiníveis.

Os métodos `insertNodeToIndex(...)`, `removeNodeFromIndex(...)`, `updateNodePartition(...)`, `insertNodeToSetWithEdgeOnCut(...)`, `getAmountOfNodesFromSet(...)` e `queryNodesFromSet(...)` delegam suas chamadas ao atributo `partitionIndex`.

Para delegar as chamadas para o atributo `cutIndex`, foram definidos os métodos `insertEdgeToCut(...)`, `queryEdgesOnCut()`, `getCutWeight()`, `calculateEdgeCut(...)` e `removeEdgeFromCut(...)`.

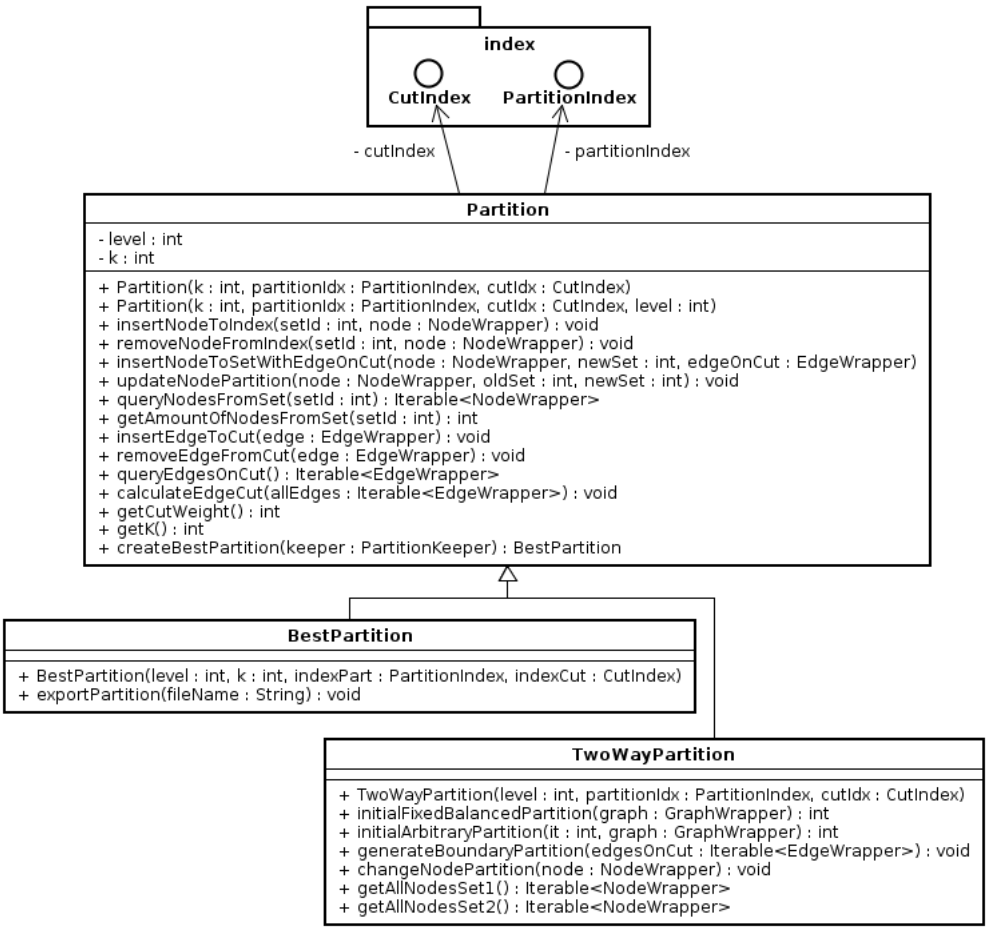


Figura 5.7: Cerne da estrutura de particionamento

O método `createBestPartition(...)` instancia um objeto da classe `BestPartition`, designado para armazenar a melhor partição alcançada até um certo momento da execução do algoritmo. Ele armazena os índices atuais de arestas no corte e vértices em cada partição, fornecido pelo atributo `keeper`. Após a obtenção dos dois índices, este método informa ao atributo `keeper` para criar novos índices, para utilização nas próximas iterações, pois os atuais devem ser mantidos para consultas futuras da melhor partição encontrada até o momento. A classe `BestPartition` também permite exportar as informações internas de particionamento para futuras consultas, através do método `exportPartition()`, para um arquivo de texto cujo nome é fornecido pelo parâmetro `fileName`.

A classe `TwoWayPartition` foi incluída neste diagrama pois ela é utilizada por vários algoritmos de particionamento *2-way*, como, KL (KERNIGHAN; LIN, 1970), BKL e Multinível (KARYPIS; KUMAR, 1995), FM (FIDUCCIA; MATTHEYSES, 1982), *Banded Diffusion* (PELLEGRINI, 2007), entre outros. Com ela, é possível criar uma partição fixa balanceada ou uma partição aleatória,

respectivamente, através dos métodos `initialArbitraryPartition(...)` e `initialFixedBalancedPartition(...)`.

Um método utilitário, chamado `changeNodePartition(...)`, é fornecido, no qual troca o vértice de partição e já atualiza os índices internos. Já os métodos `getAllNodesSet1(...)` e `getAllNodesSet2(...)` obtém os vértices da partição correspondente. Finalmente, um método que é utilizado por algoritmos multiníveis, `generateBoundaryPartition(...)`, gera uma partição somente com os vértices que estão na fronteira.

O subcomponente *index*, detalhado na Figura 5.8 possui as classes e interfaces utilizadas pelo componente *partition*. Os métodos das classes que realizam as interfaces foram omitidos para tornar o diagrama mais legível.

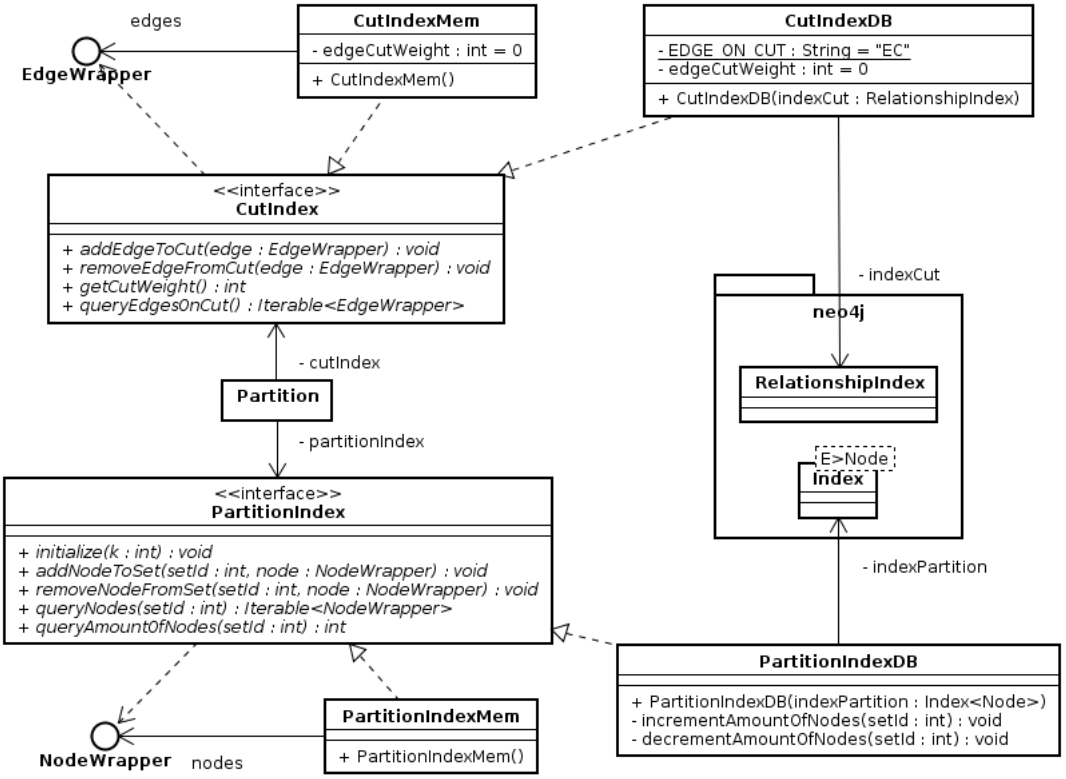


Figura 5.8: Detalhes da estrutura de índices

Os principais elementos deste componente são as interfaces `PartitionIndex` e `CutIndex`. A interface `CutIndex` define os serviços necessários para manutenção das arestas do corte, através dos seguintes métodos:

- `addEdgeToCut(...)`: adiciona uma aresta no corte atual.
- `removeEdgeFromCut(...)`: remove uma aresta do corte.

- `getCutWeight()`: recupera o valor da soma do peso das arestas que estão no corte.
- `queryEdgesOnCut()`: obtém a lista atual de arestas do corte.

Por sua vez, as classes `CutIndexMem` e `CutIndexDB` realizam a interface `CutIndex`, cada qual utilizando seus recursos de acordo com a forma de armazenamento interno dos índices. A classe `CutIndexMem` armazena, em seu atributo `edges`, uma coleção de objetos da classe `EdgeWrapper` para fornecer as informações definidas por sua interface. Já a classe `CutIndexDB` utiliza dos recursos do Neo4J, através do atributo `indexCut` do tipo `RelationshipIndex`, para fazer a indexação das arestas.

A indexação dos vértices, definida pela interface `PartitionIndex` é fornecida pelos seguintes métodos:

- `initialize(...)`: permite às subclasses fazerem as inicializações necessárias para a manipulação dos vértices.
- `addNodeToSet(...)`: adiciona um vértice em uma certa partição.
- `removeNodeFromSet(...)`: remove um vértice de uma certa partição.
- `queryNodes(...)`: obtém os vértices de uma dada partição. Este método também é utilizado para armazenar o resultado do particionamento, feito pelo método `exportPartition()` da classe `BestPartition` .
- `queryAmountOfNodes(...)`: obtém a quantidade de vértices de uma dada partição.

As classes `PartitionIndexMem` e `PartitionIndexDB` realizam a interface `PartitionIndex` para armazenamento interno dos índices de vértices por partição. A classe `PartitionIndexMem` armazena, em seu atributo `nodes`, uma coleção de objetos da classe `NodeWrapper`. Por outro lado, a classe `PartitionIndexDB` utiliza o próprio recurso de indexação interna do banco de dados Neo4J, através do atributo `indexPartition` do tipo `Index<Node>`, para fazer a indexação dos vértices.

5.4 Outras considerações sobre a arquitetura

A partir da estrutura mostrada até aqui, o pesquisador deve voltar seus esforços para a implementação do algoritmo, independentemente do ambiente de armazenamento do grafo, permitindo também a criação de outras estruturas de dados para auxiliar na execução do particionamento. Outra característica importante é a facilidade de se trocar o grafo utilizado em um mesmo algoritmo.

Vale ressaltar que a arquitetura proposta é extensível para utilização de outros bancos de dados orientados a grafos, bastando criar subclasses para as implementações específicas desejadas. Outras classes especializadas em particionamentos específicos também podem ser criadas a fim de atender a um algoritmo em particular, fornecendo meios para facilitar e acelerar seu desenvolvimento e seus testes.

6 Aplicação da Arquitetura

A partir do modelo de classes apresentado no Capítulo 5, a implementação da arquitetura foi feita utilizando a linguagem Java. A escolha desta linguagem foi fortemente influenciada pelo uso do banco de dados Neo4j, que é facilmente acessível de forma embutida dentro de um programa Java, bastando fazer a inclusão de suas bibliotecas no projeto desejado, conforme mostrado no Capítulo 3.

Com a arquitetura pronta, deu-se início à definição das classes para a implementação de quatro algoritmos clássicos de particionamento, permitindo sua utilização e validação através da execução dos experimentos, mostrando a forma de invocar um algoritmo independente da forma de armazenamento do grafo, para demonstrar ao usuário a facilidade de utilização da arquitetura.

Também serão mostradas, neste capítulo, as características dos grafos utilizados nos experimentos, bem como os resultados da execução dos algoritmos implementados.

6.1 Utilização da arquitetura

Para a validação da implementação da arquitetura proposta, os quatro algoritmos foram implementados utilizando as classes `GraphWrapper`, `TransactionControl`, `Partition` e `BestPartition`, além das interfaces `NodeWrapper`, `EdgeWrapper`, `CutIndex`, `TransactionInterface`, `PartitionIndex` e `PartitionKeeper`. Assim, a codificação de um determinado algoritmo é feita de forma genérica, deixando a decisão da escolha da forma de armazenamento do grafo para o código que executa o algoritmo.

Para facilitar o entendimento do trabalho realizado, a Figura 6.1 apresenta o diagrama com as classes necessárias para a implementação do algoritmo de Kernighan e Lin (1970), onde ainda não é especificado qual será a estrutura interna

de armazenamento do grafo.

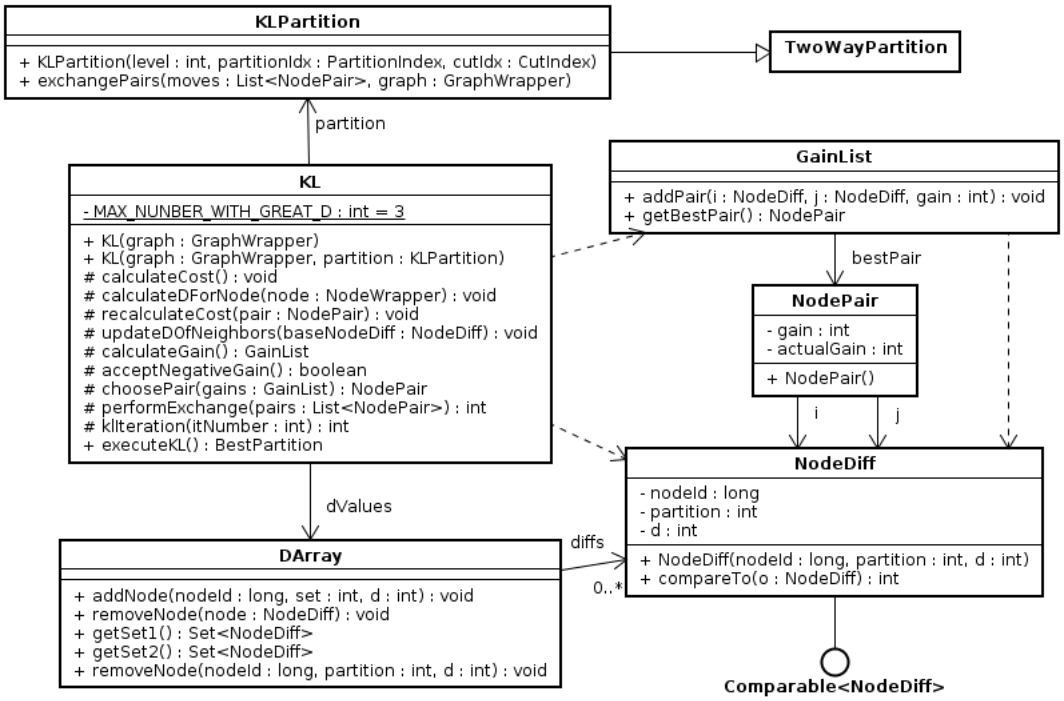


Figura 6.1: Classes do algoritmo KL

Neste diagrama, a classe `KL` utiliza as definições das interfaces `NodeWrapper` e `EdgeWrapper` e a classe abstrata `GraphWrapper` como forma de acesso aos objetos de armazenamento do grafo para fazer os cálculos de custo e ganho dos vértices, além de escolher e efetuar a troca dos pares de vértices entre as partições.

A classe `KLPartition`, que é subclasse de `TwoWayPartition`, permite fazer as operações sobre as partições, através do método `exchangePairs(...)`, que é especializado em efetivar a troca dos pares escolhidos pelo algoritmo.

A classe `KL` possui os métodos necessários para a execução dos passos do algoritmo, como cálculo e manutenção do valor de *diferença* de cada vértice, representado pela classe `NodeDiff`, cálculo e armazenamento dos ganhos, auxiliados pela classe `GainList`, que mantém os pares de vértices candidatos e já obtém o melhor par a ser trocado. A classe `NodePair` armazena o ganho de um par de vértices e a classe e `DArray` mantém ordenados os objetos da classe `NodeDiff`.

A classe `KL` possui dois construtores. Ambos possuem o parâmetro `graph`, do tipo `GraphWrapper`, que permite ao desenvolvedor definir qual grafo específico será utilizado. Assim, para a invocação deste código utilizando o grafo em memória, basta instanciar a classe `KL` especificando o objeto do grafo desejado. O

segundo construtor é utilizado pelo algoritmo BKL (KARYPIS; KUMAR, 1995), onde já existe uma partição inicial, que é recebida pelo parâmetro `partition`.

A Listagem 6.1 mostra a simplicidade de instanciar o objeto do grafo em memória principal e invocar o algoritmo através do método `executeKL()` do objeto `kl`.

Listagem 6.1: Chamada ao algoritmo KL com o grafo em memória

```
1 GraphMem graph = new GraphMem(graphFileName);
2 KL kl = new KL(graph);
3 BestPartition resultPartition = kl.executeKL();
4 resultPartition.exportPartition("kl-mem.out");
```

A linha 4 armazena o resultado da execução no arquivo `kl-mem.out`, facilitando ao usuário da arquitetura verificar posteriormente seu resultado.

Para executar o algoritmo KL utilizando o banco de dados Neo4J, basta alterar a linha 1 da Listagem 6.1 para instanciar o grafo apropriado, como mostrado na Listagem 6.2.

Listagem 6.2: Chamada ao algoritmo KL com o grafo no banco de dados

```
1 GraphDB graph = new GraphDB(graphFileName);
2 KL kl = new KL(graph);
3 BestPartition resultPartition = kl.executeKL();
4 resultPartition.exportPartition("kl-db.out");
```

Vale observar que não houve alterações nas linhas 2 e 3 das listagens 6.1 e 6.2, pois todo o código correspondente da Figura 6.1 utilizou classes genéricas e interfaces presentes na arquitetura. Desta forma, um mesmo algoritmo pode ser executado utilizando o grafo em memória principal ou em banco de dados.

Similarmente ao algoritmo KL, foram implementados outros três algoritmos para o uso e a validação da arquitetura: FM (FIDUCCIA; MATTHEYSES, 1982), *2-way* multinível (KARYPIS; KUMAR, 1995) e *Greedy-Kway* (JAIN; SWAMY; BALAJI, 2007).

A Figura 6.2 ilustra as classes criadas para a implementação do algoritmo FM, com a classe principal `FM` que utiliza a classe `Bucket` para auxiliar na manutenção dos vértices de acordo com seus ganhos e a classe `Move` para manter o vértice e seu respectivo ganho, acessados por seus métodos `get` e `set` que foram omitidos no diagrama.

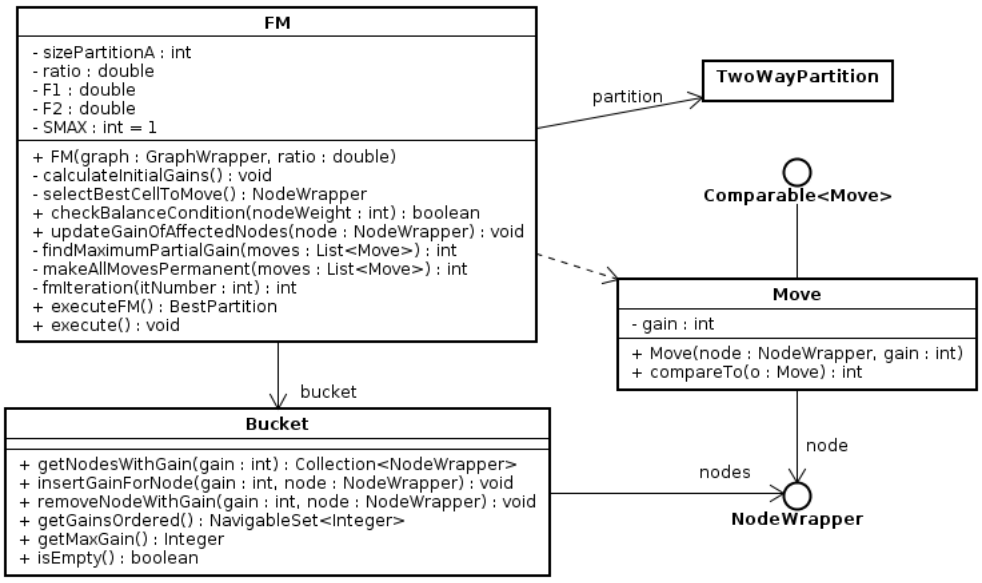


Figura 6.2: Classes do algoritmo FM

A classe `TwoWayPartition`, detalhada na Figura 5.7, provê o suporte de particionamento para a classe `FM`. As interfaces `NodeWrapper` e `EdgeWrapper` e a classe abstrata `GraphWrapper` são utilizadas pela classe `FM` para definir o acesso ao conteúdo do grafo a fim de criar um particionamento inicial, calcular os ganhos, selecionar o melhor vértice a ser movido, atualizar o ganho dos vértices afetados pelo movimento do vértice escolhido e efetivar o movimento dos vértices no particionamento. Assim, novamente, estas classes ficam independentes do tipo de armazenamento do grafo.

O código, mostrado na Listagem 6.3, utiliza o grafo no banco de dados na execução do algoritmo FM e exporta o particionamento resultante para o arquivo `fm-db.out`, similar ao código da Listagem 6.1.

Listagem 6.3: Chamada ao algoritmo FM com o grafo em banco de dados

```
1 GraphDB graph = new GraphDB(graphFileName);
2 FM fm = new FM(graph);
3 BestPartition resultPartition = fm.executeFM();
4 resultPartition.exportPartition("fm-db.out");
```

Para o algoritmo de bipartição multinível (KARYPIS; KUMAR, 1995), representado na Figura 6.3 pela classe abstrata `TwoWayMultilevel`, foram criadas duas classes para auxiliar na contração dos vértices: `CoarseHelper` e `Matching` que, a partir de um grafo original, criam um grafo contraído utilizando a técnica de emparelhamento das arestas mais leves, conforme descrito no item 1 da se-

ção 4.3, até atingir a quantidade desejada de vértices, definida pelo parâmetro `nodesInHighestLevel(...)` do método `executeMultilevel`, que neste trabalho possui o valor 500 baseado no artigo original, onde os autores sugerem que o grafo seja contraído até algumas centenas de vértices.

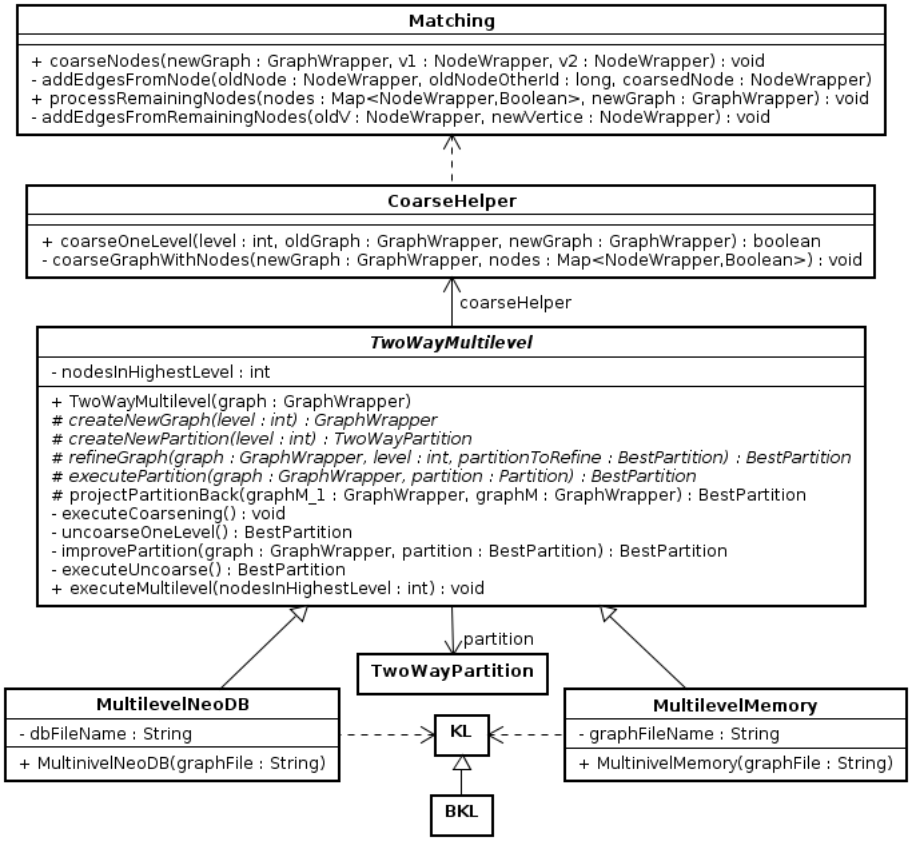


Figura 6.3: Classes do algoritmo *2-way* multinível

A classe `TwoWayMultilevel`, internamente, utiliza uma instância da classe `TwoWayPartition` para manter as informações de particionamento e possui alguns métodos abstratos, implementados por suas subclasses, `MultilevelNeoDB` e `MultilevelMemory`, para apoiar a execução do algoritmo, instanciando objetos apropriados de memória ou banco de dados.

O método `executePartition(...)`, que define o processo de particionamento, utiliza a classe `KL` para a bipartição e a classe `BKL` para o refinamento, que é feito pelo método `refineGraph(...)`. Os métodos abstratos `createNewGraph(...)` e `createNewPartition(...)` auxiliam o processo multinível à medida que é necessário avançar a um nível superior, criando o grafo e a partição correspondente.

Para auxiliar na fase de expansão do grafo, a classe `TwoWayMultilevel` pos-

sui os métodos `projectPartitionBack(...)` e `improvePartition(...)`. Por questões de simplificação, os métodos implementados pelas subclasses foram omitidos do diagrama.

Como exemplo de aplicação destas classes, a Listagem 6.4 exibe o código para invocar o algoritmo multinível utilizando o grafo no banco de dados e, finalmente, exportando o particionamento final para o arquivo `mn-db.out`.

Listagem 6.4: Chamada ao algoritmo Multinível com o grafo em banco

```
1 MultilevelNeoDB mn = new MultilevelNeoDB(graphFileName);
2 BestPartition resultPartition = mn.executeMultilevel(500);
3 resultPartition.exportPartition("mn-db.out");
```

Neste código não foi necessário criar o objeto grafo, pois a classe `MultilevelNeoDB` já o fornece de acordo com o parâmetro `graphFileName` para a execução do algoritmo utilizando o banco de dados. Isto também é válido para os novos grafos de níveis superiores.

Para suportar a execução do código multinível, foram criadas as classes que implementam o algoritmo BKL, discutido na seção 4.3.1, ilustradas na Figura 6.4.

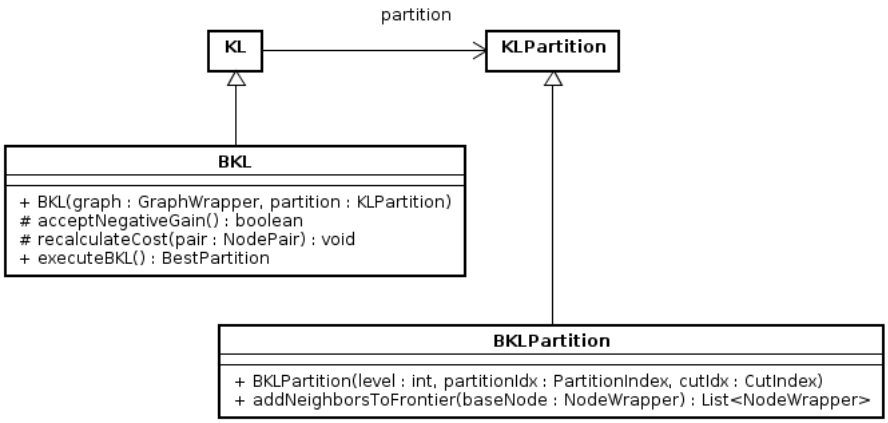


Figura 6.4: Classes do refinamento BKL

A classe `BKL` herda as características da classe `KL` adicionando dois comportamentos específicos através de seus métodos protegidos. Primeiramente, o método `recalculateCost(...)` obtém os vértices vizinhos ao vértice em questão, calcula os valores de *diferença* deles, inclui-os no particionamento (pois eles se tornaram parte da fronteira) e atualiza o valor do custo, através do método `recalculateCost(...)` da classe `KL`. Seu método `acceptNegativeGain()` sempre retornará *false*, pois, no refinamento, acredita-se que o processo de parti-

cionamento não esteja mais em um mínimo local, permitindo que somente trocas com ganhos positivos sejam aceitas.

O modelo de classes, exibido na Figura 6.3, permite que outras classes além da `MultilevelNeoDB` e `MultilevelMemory` sejam criadas, de forma que o particionamento e o refinamento sejam executados utilizando outros algoritmos. Para isto, basta reescrever adequadamente os métodos `executePartition(...)` e `refineGraph(...)`, podendo assim, haver várias combinações que o usuário desejar para execução destes passos neste ambiente multinível.

Para suportar os requisitos do algoritmo de Jain, Swamy e Balaji (2007), foram definidas as classes ilustradas na Figura 6.5.

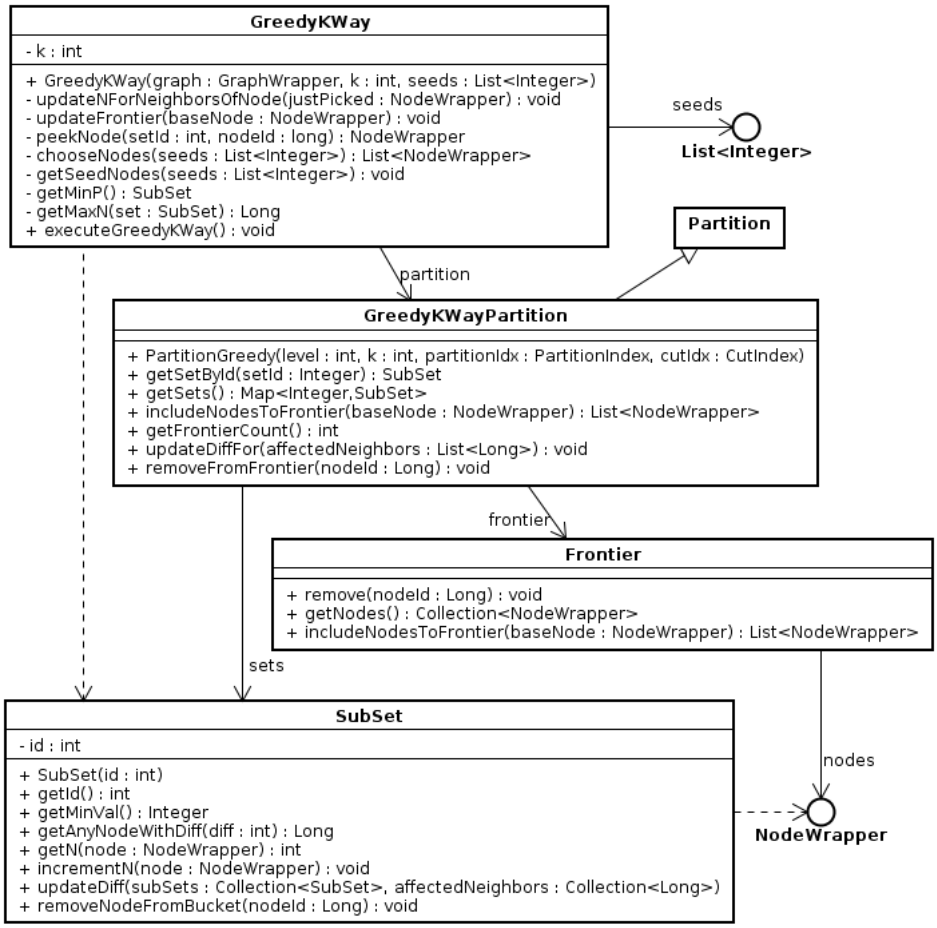


Figura 6.5: Classes do algoritmo *Greedy-KWay*

A classe `GreedyKWay` utiliza os vértices iniciais para avançar no grafo, através da fonteira que vai se formando a cada vértice consumido. Para obter o próximo vértice a ser processado, é utilizado o método `peekNode(...)`, enquanto que o método `updateFrontier(...)` adiciona novos vértices na fronteira e atualiza

e efetua os cálculos necessários.

Os métodos `getMinP(...)` e `getMaxN(...)` selecionam a próxima partição e o próximo vértice a ser adicionado a ela. Para manter as informações dos valores de N e $diff$, é utilizada a classe `SubSet` facilitando a obtenção de vértices candidatos a serem utilizados nas próximas execuções.

Para manter o particionamento, é utilizada a classe `GreedyKWayPartition`, a qual gerencia as instâncias de cada *SubSet*, bem como a fronteira que vai sendo formada conforme o algoritmo avança.

A Listagem 6.5 mostra um exemplo de uso das classes para a invocação do algoritmo *Greedy-KWay* utilizando o grafo em memória, para um particionamento em 3 subconjuntos, utilizando três números aleatórios como identificadores dos vértices iniciais.

Listagem 6.5: Chamada ao algoritmo *Greedy-KWay* com o grafo em memória

```
1 int k = 3;
2 Random rand = new Random();
3 List<Integer> seeds = Arrays.asList(new Integer [] {
4     rand.nextInt(), rand.nextInt(), rand.nextInt() });
5 GraphMem graph = new GraphMem(graphFileName);
6 GreedyKWay gkway = new GreedyKWay(graph, k, seeds);
7 BestPartition resultPartition = gkway.executeGreedyKWay();
8 resultPartition.exportPartition("g3w-db.out");
```

Com base nestes diagramas e exemplos de utilização da arquitetura, o usuário pode, tanto criar novas estruturas, quanto testar novos algoritmos de particionamento de grafos com a possibilidade de execução utilizando a memória principal ou um banco de dados orientado a grafos.

Isto favorece o aumento de implementações de algoritmos de particionamento de grafos, permitindo a realização de testes com maior eficiência sem a necessidade de alterações no código interno do algoritmo para usufluir da mudança da estrutura do grafo.

6.2 Experimentos

Esta seção mostra os resultados dos experimentos da execução do código desenvolvido, tanto da arquitetura quanto dos algoritmos.

Estes experimentos visam mostrar o impacto no desempenho dos algoritmos de particionamento implementados, considerando o armazenamento dos grafos em disco e na memória principal. Além disso, estes testes possibilitaram:

- validar a implementação da arquitetura desenvolvida;
- validar o código dos algoritmos implementados;
- medir o tempo de execução de cada algoritmo em cada forma de armazenamento.

Uma parte importante do processo experimental foi a leitura e importação dos grafos para o banco de dados. Isto foi feito através da classe `GraphFileReader`, ilustrada na Figura 6.6, que foi codificada para fazer a leitura de um arquivo texto e criar os vértices e arestas no bando de dados Neo4j, através do método `importGraph(GraphWrapper graph, String sourceFileName)`. Este método também foi utilizado para fazer a leitura dos grafos com as estruturas de dados em memória, bastando utilizar o parâmetro `graph` adequadamente.

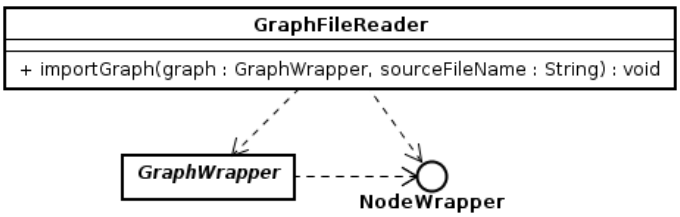


Figura 6.6: Classe para importar o grafo para o banco Neo4J

O formato dos arquivos texto, utilizados neste trabalho, seguem um padrão utilizado pelo *software* JOSTLE (WALSHAW; CROSS, 2007) e outros *softwares*, como, Chaco (HENDRICKSON, 2011) e METIS (KARYPIS; KUMAR, 2009), onde, a primeira linha possui a quantidade de vértices e arestas e, opcionalmente um código sobre as informações do grafo. Cada linha seguinte corresponde a um vértice e possui uma lista de valores, separados pelo caractere espaço, indicando seus vértices adjacentes.

Para a validação dos códigos dos algoritmos, foi utilizado um conjunto composto por nove grafos sintéticos gerados através da ferramenta GraphStream (PIGNÉ et al., 2008). Ela foi desenvolvida na linguagem Java e possui classes que geram os mais variados tipos de grafos, além de recursos para visualização e exportação de grafos como imagens e outros formatos.

Conforme ilustrado na Figura 6.7, foram criadas as subclasses da classe `BaseGenerator`, do pacote `graphstream`, para gerar os grafos, para controlar a quantidade de arestas inseridas no corte e a quantidade de vértices em cada *cluster*.

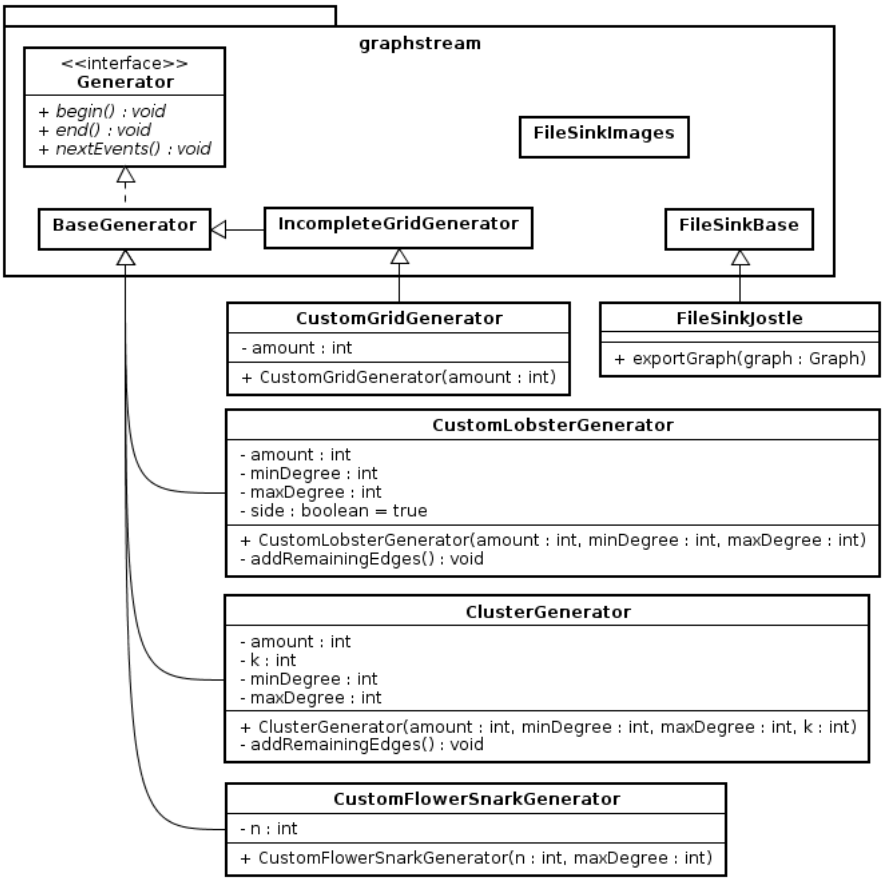


Figura 6.7: Classes para geração dos grafos sintéticos

A criação dos vértices é feita através de chamadas ao método `nextEvent()`, que cria os vértices e arestas aleatoriamente. Os métodos `begin()` e `end()` permitem fazer a inicialização e finalização do processo de geração do grafo. Estes três métodos foram omitidos das subclasses para simplificar o diagrama.

Para exportar o arquivo no final do processo de geração, com o formato de texto padrão estabelecido, foi necessário criar a subclasse `FileSinkJostle`, que herda da classe `FileSinkBase`.

Os grafos gerados são exibidos na Tabela 6.1, onde é mostrado a quantidade de vértices, a quantidade de arestas e o corte mínimo já conhecido, por serem construídos sinteticamente.

Grafo	**N^o de vértices**	**N^o de arestas**	**Menor corte**
gen_100	100	198	7
gen_500	500	951	4
gen_1000	1000	1901	4
gen_2000	2000	3829	5
flowerSnark	99	131	3
lobster	500	741	1
grid_100	112	360	12
grid_500	511	1839	27
grid_1000	1063	3947	39

Tabela 6.1: Grafos sintéticos para validação dos algoritmos implementados

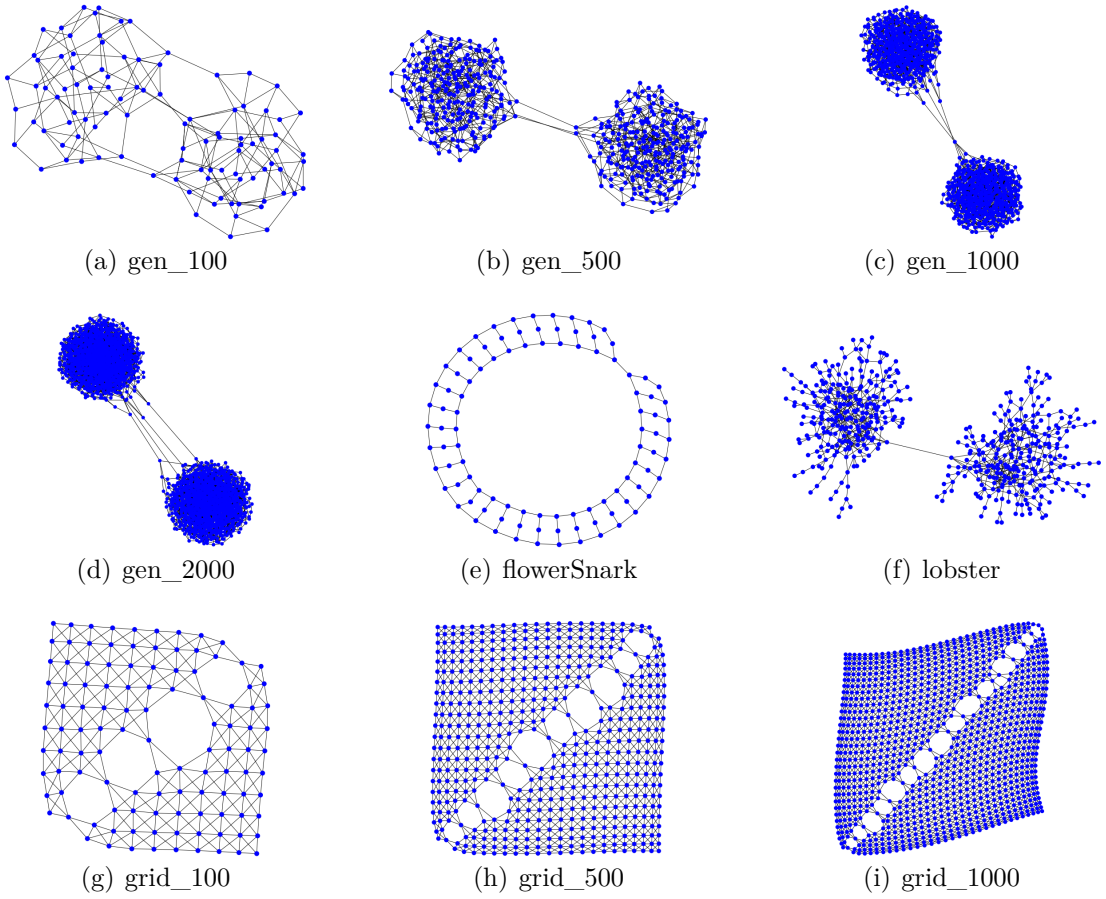
Os quatro primeiros grafos, com prefixo *gen*, foram gerados utilizando a classe `ClusterGenerator`, que permite especificar a quantidade k de conjuntos. Inicialmente são criados k vértices, um em cada *cluster*. Nas próximas iterações, novos vértices são adicionados a cada *cluster*, e são conectados aos seus vizinhos de seu próprio grupo. A quantidade de arestas que cada vértice possui é definida pelos atributos `minDegree` e `maxDegree`, que, para estes grafos, foram definidos como 3 e 6 respectivamente. Finalmente, as arestas do corte são adicionadas conectando-se todos os vértices iniciais e vértices aleatórios de diferentes *clusters*, não ultrapassando uma quantidade `maxDegree` de arestas.

O grafo *flowerSnark* foi gerado pela classe `CustomFlowerSnarkGenerator` e o grafo *lobster* foi gerado pela classe `CustomLobsterGenerator`, através de métodos que adicionaram arestas no corte até atingir a quantidade `maxDegree`.

Já os grafos de prefixo *grid* foram gerados utilizando a classe `CustomGrid-Generator`, reutilizando o processo de construção do grafo, definido em sua superclasse `IncompleteGridGenerator`.

Para caracterizar visualmente o conteúdo dos grafos da Tabela 6.1, foi utilizada a classe `FileSinkImages` do pacote `graphstream` para gerar as imagens de cada grafo, esboçados na Figura 6.8.

Para os experimentos relacionados ao consumo de memória, um outro conjunto de grafos foi utilizado, aumentando a abrangência dos experimentos. Este conjunto, publicado por Bader et al. (2013) (na categoria *Clustering Instances*),



(a) gen_100 (b) gen_500 (c) gen_1000

(d) gen_2000 (e) flowerSnark (f) lobster

(g) grid_100 (h) grid_500 (i) grid_1000

Figura 6.8: Representação visual dos grafos sintéticos da Tabela 6.1

foi utilizado devido à quantidade elevada de vértices e arestas, forçando a ocorrência de problemas na execução dos métodos de particionamento em memória. Deste segundo conjunto de grafos, somente 11 foram utilizados, conforme listados na Tabela 6.2.

De acordo com Bader et al. (2013), estes grafos possuem o formato definido por Walshaw e Cross (2007) e também possuem as seguintes características:

- possuem suas matrizes simétricas;
- não possuem arestas paralelas;
- não possuem auto-arestas;
- não possuem arestas com peso zero nem com peso infinito;
- os pesos são atributos opcionais;
- os pesos de vértices e arestas são expressos em valores inteiros.

Grafo	|V|	|A|
astro-ph	16.706	121.251
cond-mat	16.726	47.594
as-22july06	22.963	48.436
cond-mat-2003	31.163	120.029
cond-mat-2005	40.421	175.691
smallworld	100.000	499.998
G_n_pin_pout	100.000	501.198
caidaRouterLevel	192.244	609.066
cnr-2000	325.557	2.738.969
eu-2005	862.664	16.138.468
in-2004	1.382.908	13.591.473

Tabela 6.2: Grafos reais utilizados (BADER et al., 2013)

Para os vértices ou arestas que não continham pesos, foram considerados o valor unitário para a execução dos experimentos.

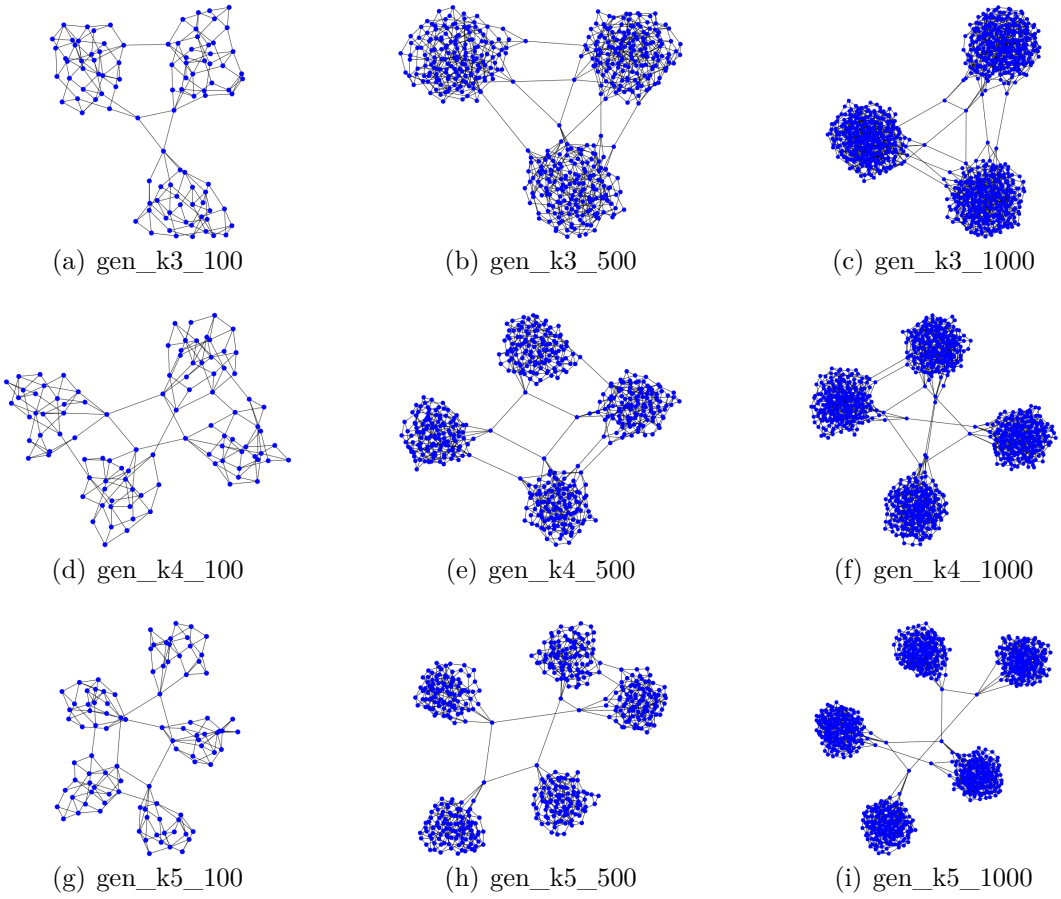
O particionamento inicial, utilizado pelos algoritmos KL, FM e pelo Multinível, foi feito utilizando o método `initialArbitraryPartition(...)` da classe `TwoWayPartition`. Isto permitiu que, através das várias execuções, se alcançasse diferentes resultados, sendo que os melhores foram utilizados neste trabalho. Já o algoritmo *Greedy K-Way* utiliza vértices iniciais aleatórios em sua execução.

Para o algoritmo *Greedy K-Way*, os experimentos foram feitos com o valor de k variando de 2 a 5, onde, para $k = 2$, foram utilizados os grafos da Tabela 6.1 e para $3 \leq k \leq 5$ foram utilizados os grafos da Tabela 6.3, que também foram gerados utilizando a classe `ClusterGenerator` com seu respectivo valor de k.

Grafo	Nº de vértices	Nº de arestas	Menor corte
gen_k3_100	100	188	5
gen_k3_500	500	946	5
gen_k3_1000	1000	1914	9
gen_k4_100	100	191	8
gen_k4_500	500	972	8
gen_k4_1000	1000	1913	9
gen_k5_100	100	195	8
gen_k5_500	500	958	7
gen_k5_1000	1000	1901	6

Tabela 6.3: Grafos sintéticos $k-way$

A representação visual de cada um dos grafos da Tabela 6.3 é ilustrada na Figura 6.9.



(a) gen_k3_100 (b) gen_k3_500 (c) gen_k3_1000

(d) gen_k4_100 (e) gen_k4_500 (f) gen_k4_1000

(g) gen_k5_100 (h) gen_k5_500 (i) gen_k5_1000

Figura 6.9: Representação visual dos grafos sintéticos da Tabela 6.3

A avaliação foi conduzida em computadores equipados com processador *Intel Core2Quad* $2,83GHz$ com 4 núcleos e cache $L2$ de $6MB$ e $2GB$ de memória RAM, executando *Ubuntu 10.04* com arquitetura de 32 *bits*.

A máquina virtual Java utilizada foi a *OpenJDK Runtime Environment* na versão 1.6.0_18, sendo configurada para executar os algoritmos utilizando as seguintes opções de memória: $-Xms1000M$ (memória inicial de 1 GB) e $-Xmx1500M$ (memória máxima de $1,5$ GB).

Com a definição dos grafos e do ambiente de execução, os algoritmos foram executados para a obtenção dos resultados.

6.3 Análise dos resultados

Os resultados obtidos das execuções dos algoritmos foram analisados utilizando duas métricas: (1) corte de arestas e (2) tempo de execução.

A principal métrica utilizada para a validação dos algoritmos implementados foi o **corte de arestas**, que é o principal objetivo da maioria dos algoritmos apresentados neste trabalho. Com ela é possível conhecer os conjuntos de vértices que estão fortemente conectados com seus vizinhos internos e fracamente conectados com seus vizinhos externos. A validação desta métrica foi feita comparando-se os resultados obtidos com os valores de corte conhecidos dos grafos sintéticos.

Já as medidas de tempo foram incluídas para mostrar o impacto da execução dos mesmos algoritmos com a utilização do grafo estando em disco, possibilitando ao usuário da arquitetura realizar, caso necessite, uma análise mais apurada de cada passo do algoritmo, pois as diferenças de tempo ficam bem mais acentuadas.

As análises foram feitas individualmente para cada um dos dois itens anteriores utilizando as informações de execução dos algoritmos implementados juntamente com os grafos definidos.

Os resultados relacionados com o **corte de arestas** foram obtidos executando vinte vezes o código de cada algoritmo *2-way* implementado (KL, FM, Multinível e *Greedy 2-Way*), para os nove grafos da Tabela 6.1, totalizando 720 execuções, cada uma utilizando um particionamento inicial aleatório. A mesma quantidade de execuções foram feitas para cada valor de $3 \leq k \leq 5$ do algoritmo *Greedy-KWay*, para cada grafo da Tabela 6.3, totalizando 540 execuções.

Apesar da aleatoriedade inicial dos algoritmos, foi possível constatar que várias execuções de todos algoritmos implementados conseguiram alcançar o objetivo do corte mínimo para os grafos sintéticos, pois pode-se validar o resultado obtido com o valor do corte conhecido para cada grafo.

Como o resultado final do particionamento depende da aleatoriedade inicial, e houve o alcance do corte mínimo pelos códigos desenvolvidos, conclui-se que as implementações atenderam as especificações apresentadas no Capítulo 4.

Para respaldar as informações geradas pelos experimentos, a outra métrica obtida foi o tempo que cada algoritmo consumiu para realizar o particionamento de cada grafo.

O tempo de execução dos algoritmos com os grafos sintéticos das Tabelas 6.1 e 6.3, tanto utilizando memória principal quanto o banco de dados Neo4J, foi relativamente baixo, não alcançando 1 segundo, pois estes são de tamanho reduzido.

Já para os grafos da Tabela 6.2, o tempo consumido para a execução dos al-

goritmos tornou-se relevante. A Tabela 6.4 apresenta os valores de tempo, em segundos, acompanhados da média aritmética e desvio padrão do tempo de execução de cada algoritmo para cada grafo, utilizando as estruturas de dados em memória principal.

Grafo	**KL**		**FM**		**Multinível**		***Greedy-2Way***	
	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ
astro-ph	3,0		5,3		3,4		1,1	
	3,2	0,09	5,5	0,16	3,5	0,24	1,15	0,01
cond-mat	1,9		3,8		2,8		0,81	
	2,0	0,45	3,9	0,08	3,1	0,29	0,83	0,01
as-22july06	2,5		5,7		**10,4**		0,8	
	2,7	0,09	6,0	0,27	11,6	0,42	0,97	0,02
cond-mat-2003	3,7		8,8		3,2		1,3	
	3,8	0,07	9,1	0,36	3,3	0,12	1,4	0,01
cond-mat-2005	5,2		11,8		4,7		1,6	
	5,4	0,09	12,1	0,53	4,9	0,11	1,7	0,01
smallworld	9,9		23,3		12,7		3,1	
	10,1	0,12	29,8	4,13	13,1	0,58	3,2	0,02
G_n_pin_pout	11,3		41,5		19,0		4,1	
	11,8	0,23	43,4	1,00	20,3	0,92	4,2	0,03
caidaRouter Level	18,9		77,9		35,4		7,1	
	19,6	0,45	131,4	26,71	41,7	2,93	7,7	0,39
cnr-2000	OoM		OoM		OoM		**13,3**	
	—	—	—	—	—	—	14,5	0,50
eu-2005	OoM		OoM		OoM		OoM	
	—	—	—	—	—	—	—	—

Tabela 6.4: Tempo de execução em memória (em segundos)

Aqui percebe-se claramente o limite da memória, destacados nas células da tabela com o termo *Out of Memory* (OoM), onde a execução destes algoritmos é impedida devido ao tamanho os grafos.

Estes resultados demonstram a superioridade de desempenho do algoritmo *Greedy-KWay* para o biparticionamento, conforme descrito por Jain, Swamy e Balaji (2007). Outra característica obtida com a implementação atual é a particularidade deste algoritmo *Greedy-KWay* em conseguir executar o particionamento com o grafo *cnr-2000* (consumindo 13,3 segundos). Isto foi alcançado devido ao baixo consumo de memória necessária para manter as informações da fronteira.

Um aspecto demonstrado por Karypis e Kumar (1995) é que, mesmo com a execução extra de contração e expansão feita pelo algoritmo Multinível, seu tempo de execução não é tão elevado, pois é obtido um particionamento de boa qualidade com o grafo contraído, evitando que haja uma busca por vértices distantes da

fronteira nos grafos mais refinados. Apesar disto, a contração pode criar grafos de níveis elevados, como é o caso do grafo *as-22july06*, que chegou ao nível 386 para reduzir o grafo original a 500 vértices (conforme definido na chamada do método `executeMultilevel(...)` da linha 2 da Listagem 6.4) e consumindo 10,4 segundos (considerado elevado em relação às outras execuções com o mesmo grafo). Este comportamento ocorre quando a contração cria *multinodes* de grau elevado, bloqueando a contração das outras arestas conectadas nele, levando a uma quantidade maior de níveis.

A parte mais demorada da execução dos experimentos corresponde às execuções dos algoritmos utilizando os grafos armazenados no banco de dados Neo4J. A Tabela 6.5 mostra os tempos de cada algoritmo, em segundos, para cada grafo, utilizando as estruturas de dados no banco de dados Neo4J, bem como seus respectivos valores de média aritmética e desvio padrão do tempo de execução.

Grafo	**KL**		**FM**		**Multinível**		***Greedy-2Way***	
	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ
astro-ph	264		117		88		26	
	302	25,6	131	11,1	89	1,5	32,4	9,1
cond-mat	119		80		115		14	
	124	3,1	88	3,8	124	7,3	18,8	2,7
as-22july06	188		79		**431**		24	
	222	18,3	96	10,4	483	46,7	24,7	1,0
cond-mat-2003	398		173		123		38	
	430	21,8	202	16,1	128	2,8	46,0	5,6
cond-mat-2005	1.059		231		193		60	
	1.163	63,5	277	30,1	196	2,4	62,9	3,2
smallworld	948		858		1.114		126	
	987	29,0	1.014	151	1.345	144	133	6,8
G_n_pin_pout	1.062		664		1.748		150	
	1.123	42,2	745	42,3	2.183	273	162	10,0
caidaRouter Level	1.667		1.522		2.397		222	
	1.776	85,2	2.053	280	2.625	304	233	16,4
cnr-2000	4.160		2.375		3.614		330	
	4.392	186	2.534	172	3.971	273	339	16,6
eu-2005	18.171		8.519		7.256		523	
	18.378	405	9.271	284	7.740	225	562	19,6

Tabela 6.5: Tempo de execução utilizando o banco de dados Neo4j (em segundos)

A principal característica visível nestes resultados demonstra que a execução dos algoritmos utilizando o grafo no Neo4J não acarretou problemas de memória, mesmo utilizando os grafos maiores, porém, conforme esperado, o tempo foi consideravelmente maior. Um ponto importante observado neste experimento é a

utilização da classe `TransactionControl`, discutida no Capítulo 5, que permitiu efetivar as alterações feitas no grafo quando um limite de operações fosse alcançado, liberando memória para os passos seguintes dos algoritmos. A falta deste recurso provoca problemas de memória, mesmo com grafos menores.

Aqui é importante ressaltar que o algoritmo Multinível cria um novo grafo para cada nível de contração, novamente acentuando o tempo gasto para o grafo *as-22july06*.

Os resultados apresentados neste capítulo demonstram que a arquitetura proposta pode ser utilizada para facilitar a implementação de algoritmos de particionamento de grafos e que, apesar da grande diferença entre o tempo de execução em memória e em banco de dados, a utilização do banco permite a execução com grafos muito maiores.

Outro ponto importante é que as implementações alcançaram um bom nível de qualidade de particionamento, comparados com os valores de cortes definidos nos grafos gerados.

A utilização da arquitetura permitiu que a implementação fosse feita de maneira padronizada em relação ao acesso às informações do grafo e do particionamento, possibilitando a criação de um único conjunto de classes para um mesmo algoritmo trabalhar com os dados na memória ou no banco de dados Neo4J. Este é o seu objetivo, permitindo aos seus usuários preocuparem-se com o algoritmo, e não com a forma de armazenamento do grafo.

7 Conclusão

Este trabalho apresentou uma arquitetura de *software* que facilita a implementação de algoritmos de particionamento de grafos por parte dos usuários, além de permitir que um mesmo algoritmo seja executado utilizando diferentes formas de armazenamento e representação do grafo processado, sem a necessidade de alterações no código interno do algoritmo.

Para prover as informações necessárias de um grafo para a execução dos algoritmos, a arquitetura define uma camada de abstração facilitando e padronizando o acesso a este grafo, deixando a implementação do algoritmo independente de como as informações estão armazenadas.

A arquitetura também provê um serviço de manutenção do particionamento, onde, a partir da definição da quantidade de partições, o usuário pode inserir e remover vértices de cada partição de forma padronizada e prática.

Assim, os pesquisadores envolvidos na área de particionamento podem realizar seus testes com maior eficiência sem a necessidade de alterações no código do algoritmo para utilizar outras formas de armazenamento. Isto também permite uma análise pontual de qualquer fragmento de um algoritmo, pois uma simples troca do grafo da memória para o banco de dados pode levantar informações de gargalos em certas partes da implementação do algoritmo, que não eram perceptíveis com o uso da memória.

Este trabalho favorece o aumento de implementações de algoritmos de particionamento de grafos, pois os usuários, consultando os diagramas e exemplos de utilização da arquitetura, podem criar novas estruturas e testar novos algoritmos de particionamento de grafos com a possibilidade de execução, utilizando a memória principal ou um banco de dados orientado a grafos.

Os quatro algoritmos implementados permitiram validar a estrutura da arquitetura, que se mostrou adequada para um biparticionamento ou para o particio-

namento *k-way* através do método *Greedy K-Way*.

Para a validação da implementação dos algoritmos, foram utilizados os grafos sintéticos, criados através de classes especializadas da ferramenta *GraphStream* e para o teste de *stress* de memória foram utilizados outros grafos de maior tamanho disponibilizados pela comunidade.

Apesar deste trabalho não oferecer um novo algoritmo de particionamento, foi demonstrado a validade da arquitetura e dos algoritmos implementados, mostrando também o tempo gasto utilizando os grafos em memória e em banco de dados.

7.1 Contribuições

A principal contribuição deste trabalho foi a elaboração e construção de uma arquitetura de *software* que facilita a implementação de algoritmos de particionamento de grafos, através da padronização do acesso ao grafo e manutenção do particionamento, para seu uso em memória ou no banco de dados orientados a grafos Neo4J. Esta solução foi modelada a partir das necessidades encontradas nas implementações dos algoritmos utilizados para a realização dos testes. A arquitetura também permite que sejam definidas classes para acesso a outros bancos de dados, facilitando sua expansão para abranger uma quantidade maior de usuários.

Outra contribuição foi a comparação da execução dos mesmos algoritmos utilizando os mesmos grafos tanto em memória, quanto no banco de dados Neo4J. Esta comparação é importante, pois o volume de informações que são processadas ultrapassa os limites de memória disponíveis, fazendo com que os algoritmos necessitem trabalhar diretamente com os dados em disco.

O estudo e a utilização do Neo4J também possui sua relevância, pois, além de ser uma ferramenta relativamente nova, permite efetuar o armazenamento e manipulação do grafo dentro do código Java, através de sua API, aproveitando seus recursos de relacionamentos, que evitam o uso de *joins* caros presentes nos bancos relacionais.

Vale ressaltar que as implementações dos algoritmos podem ser utilizadas e adaptadas em problemas de classificação de dados, bem como em vários outros problemas da vida real, como redes de iteração de proteínas, redes sociais, segmentação de imagens entre outras.

Uma última contribuição é o estudo e sumarização da bibliografia dos algoritmos de particionamento, pois, apesar de haver vários algoritmos clássicos, a síntese aqui apresentada facilita o desenvolvimento de trabalhos futuros na área.

7.2 Trabalhos futuros

Para dar continuidade a este trabalho, existem várias propostas para trabalhos futuros que podem ser consideradas.

Do ponto de vista da arquitetura, pode-se incorporar classes para permitir a utilização de outros bancos de dados orientados a grafos, aumentando o alcance da solução na comunidade, além de criar classes especializadas de particionamento, específicas para certos tipos de algoritmos que utilizam diferentes técnicas para alcançar os seus objetivos.

Além disso, podem ser criados novos recursos na arquitetura para suportar o particionamento em um ambiente distribuído para tratamento de grafos dinâmicos, ou seja, que possuem inserções e remoções tanto de vértices quanto de arestas.

Também é importante implementar outros algoritmos de particionamento a fim de criar novos recursos na arquitetura para aumentar sua abrangência de utilização, bem como criar recursos de instrumentação do grafo para obtenção de estatísticas de acessos e gravações realizadas por um algoritmo, a fim de melhorar seu desempenho.

Com relação aos algoritmos espectrais, pode-se implementar os cálculos relativos à matriz laplaciana ou os métodos iterativos utilizando diretamente o grafo ao invés da matriz de adjacência, evitando que todo o grafo seja colocado em memória.

Outro aspecto a ser considerado para a continuação natural deste trabalho, é a criação de novos algoritmos de particionamento, inclusive utilizando técnicas híbridas a partir dos métodos já implementados.

Finalmente, há a possibilidade de incorporação da arquitetura desenvolvida bem como de algoritmos de particionamento dentro do banco de dados Neo4J ou outro banco de dados orientado a grafos.

Referências Bibliográficas

ABREU, N. de. Teoria espectral dos grafos: um híbrido entre a álgebra linear ea matemática discreta e combinatória com origens na química quântica. *TEMA-Tendências em Matemática Aplicada e Computacional*, v. 6, n. 1, p. 1–10, 2011.

ALDECOA, R.; MARÍN, I. Deciphering network community structure by surprise. *CoRR*, abs/1105.2459, 2011.

BADER, D. A. et al. (Ed.). *Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*, v. 588 de *Contemporary Mathematics*, (Contemporary Mathematics, v. 588). [S.l.]: American Mathematical Society, 2013.

BECKER, B. et al. Greedy_IIP: Partitioning large graphs by greedy iterative improvement. In: *DSD*. [S.l.]: IEEE Computer Society, 2001. p. 54–61.

BLONDEL, V. D. et al. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics*, p. 10008, jul. 25 2008.

BRANDES, U. et al. On modularity clustering. *IEEE Trans. Knowl. Data Eng*, v. 20, n. 2, p. 172–188, 2008.

BRANDES, U.; ERLEBACH, T. (Ed.). *Network Analysis: Methodological Foundations*, v. 3418 de *Lecture Notes in Computer Science*, (Lecture Notes in Computer Science, v. 3418). [S.l.]: Springer, 2005.

CONTRIBUTORS, G. P. *GSL - GNU Scientific Library - GNU Project - Free Software Foundation (FSF)*. 2010. Http://www.gnu.org/software/gsl/. Disponível em: <http://www.gnu.org/software/gsl/>.

DELLING, D. et al. Graph partitioning with natural cuts. In: *IPDPS*. [S.l.]: IEEE, 2011. p. 1135–1146.

DELLING, D. et al. Orca reduction and contraction graph clustering. In: GOLDBERG, A. V.; ZHOU, Y. (Ed.). *Algorithmic Aspects in Information and Management, 5th International Conference, AAIM 2009, San Francisco, CA, USA, June 15-17, 2009. Proceedings*. [S.l.]: Springer, 2009. (Lecture Notes in Computer Science, v. 5564), p. 152–165.

DONGARRA, J.; DOWNEY, A.; SEYMOUR, K. *JLAPACK: Java translation of the ARPACK library.* 2013. `http://icl.cs.utk.edu/f2j/`. Acessado em 12/02/2013.

DUTOT, A.; OLIVIER, D.; SAVIN, G. conference proceeding, *Centroids: a decentralized approach*. set. 2011.

ELKAN, C. Using the triangle inequality to accelerate k-means. In: FAWCETT, T.; MISHRA, N. (Ed.). *Machine Learning, Proceedings of the Twentieth International Conference (ICML 2003), August 21-24, 2003, Washington, DC, USA*. [S.l.]: AAAI Press, 2003. p. 147–153.

FIDUCCIA, C.; MATTHEYSES, R. A linear-time heuristic for improving network partitions. *Design Automation, 1982. 19th Conference on*, p. 175–181, June 1982.

FIEDLER, M. Algebraic connectivity of graphs. *cmj*, v. 23, p. 298–305, 1973.

FORTUNATO, S. Community detection in graphs. *Physics Reports*, v. 486, n. 3–5, p. 75 – 174, 2010.

FORTUNATO, S.; BARTHELEMY, M. *Resolution limit in community detection*. jul. 14 2006.

FREEMAN, L. C. Centrality in social networks: Conceptual clarification. *Social Networks*, v. 1, p. 215–239, 1979.

GEHWEILER, J.; MEYERHENKE, H. A distributed diffusive heuristic for clustering a virtual P2P supercomputer. In: *IPDPS Workshops*. [S.l.]: IEEE, 2010. p. 1–8.

GODSIL, C.; ROYLE, G. *Algebraic Graph Theory*. [S.l.]: Springer, 2001. 464 p.

HARTUV; SHAMIR. A clustering algorithm based on graph connectivity. *IPL: Information Processing Letters*, v. 76, 2000.

HENDRICKSON, B. Chaco. In: PADUA, D. A. (Ed.). *Encyclopedia of Parallel Computing*. [S.l.]: Springer, 2011. p. 248–249.

HERNÁNDEZ, V. et al. *A Survey of Software for Sparse Eigenvalue Problems*. [S.l.], jun. 2007.

HICKLIN, J. et al. *JAMA : A Java Matrix Package*. 2013. `http://math.nist.gov/javanumerics/jama/`. Acessado em 12/02/2013.

HOGBEN, L. Spectral graph theory and the inverse eigenvalue problem of a graph. *Electronic Journal of Linear Algebra*, v. 14, p. 12–31, Jan 2005.

JAIN, S.; SWAMY, C.; BALAJI, K. *Greedy Algorithms for k-way Graph Partitioning*. 2007.

KANNAN; VEMPALA; VETTA. On clusterings: Good, bad and spectral. *JACM: Journal of the ACM*, v. 51, 2004.

KARYPIS, G.; KUMAR, V. *A fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs*. Minneapolis, MN 55414, maio 1995.

KARYPIS, G.; KUMAR, V. *MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0*. 2009. `http://www.cs.umn.edu/~metis`.

KEILHAUER, A. et al. *JLinAlg: An open source and easy-to-use Java library for linear algebra.* 2013. `http://jlinalg.sourceforge.net/`. Acessado em 12/02/2013.

KERNIGHAN, B. W.; LIN, S. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, v. 49, p. 291–307, 1970 1970.

LOVÁSZ, L. Random walks on graphs: A survey. *Combinatorics, Paul Erdos is Eighty*, János Bolyai Mathematical Society, v. 2, n. 1, p. 1–46, 1993.

LUXBURG, U. von. A tutorial on spectral clustering. *CoRR*, abs/0711.0189, 2007.

MENÉNDEZ, H.; CAMACHO, D. A genetic graph-based clustering algorithm. In: YIN, H.; COSTA, J.; BARRETO, G. (Ed.). *Intelligent Data Engineering and Automated Learning - IDEAL 2012*. [S.l.]: Springer Berlin Heidelberg, 2012, (Lecture Notes in Computer Science, v. 7435). p. 216–225.

MERRIS, R. Laplacian matrices of graphs: A survey. *Linear Algebra and its Applications*, v. 197/198, p. 143–176, jan. 1994. Second Conference of the International Linear Algebra Society (ILAS) (Lisbon, 1992).

MEYERHENKE, H.; MONIEN, B.; SCHAMBERGER, S. Accelerating shape optimizing load balancing for parallel FEM simulations by algebraic multigrid. In: *IPDPS*. [S.l.]: IEEE, 2006.

MEYERHENKE, H.; SAUERWALD, T. Beyond good partition shapes: An analysis of diffusive graph partitioning. *Algorithmica*, v. 64, n. 3, p. 329–361, 2012.

NASCIMENTO, M. C.; CARVALHO, A. C. de. Spectral methods for graph clustering – a survey. *European Journal of Operational Research*, v. 211, n. 2, p. 221 – 231, 2011.

NETO, P. O. B. *Grafos: teoria, modelos, algoritmos*. São Paulo SP: E. Blucher, 1996.

NEWMAN, M. E. J. Community detection and graph partitioning. *CoRR*, abs/1305.4974, 2013.

NEWMAN, M. E. J.; GIRVAN, M. *Finding and evaluating community structure in networks*. ago. 11 2003. 026113 p.

NG, A. Y.; JORDAN, M.; WEISS, Y. On spectral clustering: Analysis and an algorithm. In: DIETTERICH, T. G.; BECKER, S.; GHAHRAMANI, Z. (Ed.). *Advances in Neural Information Processing Systems 14*. Cambridge, MA: MIT Press, 2002.

OSIPOV, V.; SANDERS, P. n-level graph partitioning. *CoRR*, abs/1004.4024, 2010.

PELLEGRINI, F. A parallelisable multi-level banded diffusion scheme for computing balanced partitions with smooth boundaries. In: KERMARREC, A.-M.; BOUGÉ, L.; PRIOL, T. (Ed.). *Euro-Par*. [S.l.]: Springer, 2007. (Lecture Notes in Computer Science, v. 4641), p. 195–204.

PIGNÉ, Y. et al. Graphstream: A tool for bridging the gap between complex systems and dynamic graphs. *CoRR*, abs/0803.2093, 2008.

ROBINSON, I.; WEBBER, J.; EIFREM, E. *Graph Databases*. [S.l.]: O'Reilly Media, 2013. 224 p.

RUAN, J.; ZHANG, W. An efficient spectral algorithm for network community discovery and its applications to biological and social networks. In: *ICDM*. [S.l.]: IEEE Computer Society, 2007. p. 643–648.

SCHAEFFER, S. E. Stochastic local clustering for massive graphs. In: HO, T. B.; CHEUNG, D. W.-L.; LIU, H. (Ed.). *Advances in Knowledge Discovery and Data Mining, 9th Pacific-Asia Conference, PAKDD 2005, Hanoi, Vietnam, May 18-20, 2005, Proceedings*. [S.l.]: Springer, 2005. (Lecture Notes in Computer Science, v. 3518), p. 354–360.

SCHAEFFER, S. E. Graph clustering. *Computer Science Review*, v. 1, n. 1, p. 27 – 64, 2007.

SEIDMAN, S. B. Network structure and minimum degree. *Social Networks*, v. 5, n. 3, p. 269–287, 1983.

SHI, J.; MALIK, J. Normalized cuts and image segmentation. *IEEE Conf. Computer Vision and Pattern Recognition*, jun. 1997.

THE NEO4J TEAM. *The Neo4j Manual*. 1.9.2. ed. [S.l.], 2013. Disponível em: <http://docs.neo4j.org/pdf/neo4j-manual-stable.pdf>.

WALSHAW, C.; CROSS, M. JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview. In: MAGOULES, F. (Ed.). *Mesh Partitioning Techniques and Domain Decomposition Techniques*. [S.l.]: Civil-Comp Ltd., 2007. p. 27–58. (Invited chapter).

WHITE, S.; SMYTH, P. A spectral clustering approach to finding communities in graph. In: *SDM*. [S.l.: s.n.], 2005.