Ministério da
Ciência, Tecnologia
e Inovação

sid.inpe.br/mtc-m19/2013/06.06.13.36-TDI

ARMAZENAMENTO E PROCESSAMENTO DE GRANDES GRAFOS EM BANCOS DE DADOS GEOGRÁFICOS

Eric Silva Abreu

Dissertação de Mestrado em Computação Aplicada, orientada pelos Drs. Luciano Vieira Dutra, Sérgio Rosim, e João Ricardo de Freitas Oliveira, aprovada em 17 de maio de 2013.

URL do documento original:
<http://urlib.net/8JMKD3MGP7W/3E8T5CB>

INPE
São José dos Campos
2013

PUBLICADO POR:

Instituto Nacional de Pesquisas Espaciais - INPE
Gabinete do Diretor (GB)
Serviço de Informação e Documentação (SID)
Caixa Postal 515 - CEP 12.245-970
São José dos Campos - SP - Brasil
Tel.:(012) 3208-6923/6921
Fax: (012) 3208-6919
E-mail: pubtc@sid.inpe.br

Ministério da
Ciência, Tecnologia
e Inovação

sid.inpe.br/mtc-m19/2013/06.06.13.36-TDI

ARMAZENAMENTO E PROCESSAMENTO DE GRANDES GRAFOS EM BANCOS DE DADOS GEOGRÁFICOS

Eric Silva Abreu

Dissertação de Mestrado em Computação Aplicada, orientada pelos Drs. Luciano Vieira Dutra, Sérgio Rosim, e João Ricardo de Freitas Oliveira, aprovada em 17 de maio de 2013.

URL do documento original:
<http://urlib.net/8JMKD3MGP7W/3E8T5CB>

INPE
São José dos Campos
2013

Dados Internacionais de Catalogação na Publicação (CIP)

Abreu, Eric Silva.
Ab86a Armazenamento e processamento de grandes grafos em bancos de dados geográficos / Eric Silva Abreu. – São José dos Campos : INPE, 2013.
xxiv + 87 p. ; (sid.inpe.br/mtc-m19/2013/06.06.13.36-TDI)

Dissertação (Mestrado em Computação Aplicada) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2013.
Orientadores : Drs. Luciano Vieira Dutra, Sérgio Rosim, e João Ricardo de Freitas Oliveira.

1. grafos. 2. bancos de dados geográficos. 3. cache. 4. TerraLib. I.Título.

CDU 004.652

Aprovado (a) pela Banca Examinadora em cumprimento ao requisito exigido para obtenção do Título de *Mestre* em

Computação Aplicada

Dra. Lúbia Vinhas

Presidente / INPE / São José dos Campos - SP

Dr. Luciano Vieira Dutra

Orientador(a) / INPE / SJCampos - SP

Dr. Sergio Rosim

Orientador(a) / INPE / São José dos Campos - SP

Dr. João Ricardo de Freitas Oliveira

Orientador(a) / INPE / SJCampos - SP

Dr. Luiz Antonio Nogueira Lorena

Membro da Banca / INPE / São José dos Campos - SP

Dr. Laércio Massaru Namikawa

Membro da Banca / INPE / São José dos Campos - SP

Dr. Antônio Henrique Correia

Convidado(a) / DSG/EB / Brasília - DF

Este trabalho foi aprovado por:

( ) maioria simples

( ) unanimidade

Aluno (a): *Eric Silva Abreu*

"O homem converte-se aos poucos naquilo que acredita poder vir a ser. Se me repetir incessantemente a mim mesmo que sou incapaz de fazer determinada coisa, é possível que isso acabe finalmente por se tornar verdade. Pelo contrário, se acreditar que a posso fazer, acabarei garantidamente por adquirir a capacidade para a fazer, ainda que não a tenha num primeiro momento".

Mahatma Gandhi
em *"The Words of Gandhi"*

A meus pais Cláudio e Márcia, à minha esposa Rita Helena e a meus filhos Anna Júlia e Theo

AGRADECIMENTOS

A Deus por estar sempre comigo.

Ao Instituto Nacional de Pesquisas Espaciais - INPE, pela oportunidade de estudo.

Aos orientadores Dr. Luciano Vieira Dutra, Dr. Sérgio Rosim, Dr. João Ricardo de Freitas Oliveira, pelas orientações, ensinamentos e confiança.

Aos docentes do Curso de Pós-Graduação em Computação Aplicada (CAP) e funcionários da Divisão de Processamento de Imagens (DPI), que auxiliaram na minha formação tornando possível o desenvolvimento deste trabalho. Em especial ao Dr. Gilberto Ribeiro de Queiroz.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo auxílio financeiro.

À Fundação de Ciência, Aplicações e Tecnologias Espaciais - FUNCATE, por permitir meu aprimoramento tanto técnico como científico.

A meus pais Cláudio de Abreu e Márcia Conceição Silva Abreu, pelo amor, incentivo e sacrifícios aos meus sonhos. Obrigado por vocês sempre acreditarem em mim.

À minha esposa Rita Helena, pelo amor, carinho e compreensão nos momentos que não pude me dedicar totalmente a você. Você foi fundamental nesta etapa da minha vida.

A meus filhos e razão da minha vida, Anna Júlia e Theo. Se hoje tenho força e ânimo para acordar todas as manhas é graças a vocês.

Aos amigos do curso, da FUNCATE e tantos outros que de formas diferentes contribuíram para a conclusão desta dissertação.

RESUMO

Em sistemas de informações geográficas, o conceito de rede podem representar, por exemplo, as informações associadas a serviços de utilidade pública, como água, luz e telefone; e também a redes relativas a bacias hidrográficas e rodovias. As informações espaciais são usualmente armazenadas em forma de grafo que contêm informações sobre recursos que fluem entre localizações geográficas distintas. Essas redes podem ser tão grandes e densas que tornam sua manipulação inviável, seja por falta de memória RAM disponível no computador ou por falta de ferramentas apropriadas. Este trabalho apresenta uma metodologia que permite o armazenamento e manipulação de grandes grafos em banco de dados geográficos através da definição de um conjunto de tabelas relacionais que descrevem os metadados do grafo e a definição de uma política de *cache* para otimizar o acesso aos dados. Como não existe nenhuma definição formal de como se deve definir o modelo de grafo em um banco de dados, é feito um estudo sobre as diversas tecnologias de mercado que se utilizam desse conceito. O modelo proposto é flexivel o suficiente para que possa ser utilizado por diferentes tipos de aplicações permitindo o processamento de grandes bases de dados. Os grafos abordados nesse trabalho são aqueles que podem ser espacialmente definidos e utilizados para representar modelos hidrológicos, fluxos e regiões de adjacência.

STORAGE AND PROCESSING OF LARGE GRAPHS IN GEOGRAPHICAL DATABASES

ABSTRACT

In geographic information systems, the concept of a network represents the information associated with public utility services such as water, electricity and telephone, as well as networks for watershed and highways. Spatial information is usually stored as a graph that contain information about resources that flow between different geographical locations. These networks can be so large and dense that handling them is infeasible, either due to the lack of available RAM on your computer or appropriate tools. This dissertation presents a methodology that allows the storage and manipulation of large graphs in a geographic database by defining a set of relational tables that describe the metadata of the graph and the definition of a cache policy in order to optimize data access. As there is no formal definition of how to define the graph model in a database, a study is made of the various technologies in the market that use this concept. The proposed model is flexible enough so that it can be used for different types of applications allowing the processing of large databases. The graphs discussed in this paper are those that can be spatially defined and used to represent hydrological models, flows and regions adjacency.

LISTA DE FIGURAS

Pág.

LISTA DE TABELAS

Pág.

LISTA DE ABREVIATURAS E SIGLAS

API	–	Application Programming Interface
BFS	–	Breadth First Search
BGL	–	The Boost Graph Library
DDL	–	Data Definition Language
DML	–	Data Manipulation Language
FIFO	–	First In First Out
G-GCS	–	Geographical Aware Graph-based Coupling Structure
GML	–	The Graph Modelling Language
LDD	–	Local Drain Directions
LRU	–	Least Recently Used
MBR	–	Minimum Bounding Rectangle
MNT	–	Modelo Numérico de Terreno
OO	–	Orientado a Objeto
PL/SQL	–	Procedural Language/Structured Query Language
RAG	–	Region Adjacency Graphs
SGBD	–	Sistema Gerenciador de Banco de Dados
SIG	–	Sistema de Informação Geográfica
SQL	–	Structured Query Language
TSP	–	Traveling Salesperson Problem

SUMÁRIO

Pág.

1 INTRODUÇÃO

A necessidade de se relacionar e conectar objetos ou pessoas é uma demanda que ocorrem frequentemente em diversas áreas. Serviços de utilidade pública (água, luz e telefone), redes de drenagem, malhas viárias, etc., têm em comum a característica de possuírem uma relação entre seus componentes e poderem ser representadas por grafos. Os grafos são estruturas que representam um conjunto de dados juntamente com as ligações existentes entre seus elementos. Cada objeto (ex: central telefônica, transformador de rede elétrica, estação de tratamento de água) pode ser representado como um vértice desse grafo e as ligações entre esses objetos (ex: cabos telefônicos, fios de alta tensão, canos de água) representado por arestas.

Este tipo de estrutura pode ser utilizada para modelar diversos problemas. Neste trabalho específico iremos abordar os problemas cuja localização espacial é uma componente sempre presente. Nestes grafos, além das associações entre os objetos, existe uma associação explícita com o espaço, dada a localização dos objetos; essas informações geográficas dos grafos são dadas através de coordenadas espaciais. Ao armazenarmos esse tipo de informação em banco de dados geográficos é possível utilizar-se de uma série de operações que facilitam a manipulação desses dados.

Atualmente existem disponíveis várias aplicações ou bibliotecas computacionais para se trabalhar com grafos, sejam bibliotecas escritas em C++ como *The Boost Graph Library - BGL* (SIEK et al., 2002), *LEDA* (MEHLHORN; NäHER, 1995) ou bancos de dados relacionais, espaciais ou mesmo orientados a grafos, entre outras aplicações. O foco deste trabalho não se concentra na implementação de algoritmos de operações sobre grafos, mas sim em como armazenar e manipular grandes quantidades de dados fazendo um uso inteligente da memória RAM do computador.

Diversas aplicações abordam os problemas de armazenamento e representação das estruturas de grafos, tais como:

- Sistemas de Informações Geográficas: acessam os dados a partir de um repositório e definem seus modelos de grafos;
- Sistemas Gerenciadores de Banco de Dados: cada gerenciador define um conjunto de tabelas para representar seu modelo de grafo;
- Bancos de Dados Orientados a Grafos: os dados são armazenados levando-se em conta o relacionamento dos objetos.

Observa-se que já existem disponíveis inúmeras soluções, seja para seu armazenamento ou para operações sobre grafos. Uma abordagem interessante seria criar uma camada intermediária (*middleware*) que seja capaz de se comunicar com as diversas fontes de dados e também interagir com as bibliotecas de grafos que fornecem os algoritmos de percorrimento.

1.1 Motivação

Em Rosim (2008) é proposto um modelo para a representação dos fluxos locais de Modelos Hidrológicos que elimina o acoplamento entre o modelo de representação do terreno e o armazenamento. Este desacoplamento é feito utilizando a *Geographical Aware Graph-based Coupling Structure* - G-GCS, que utiliza estruturas de grafos para a representação dos fluxos locais (Fig. 1.1).

A aplicação TerraHidro (ROSIM et al., 2003) é uma ferramenta de Modelagem Hidrológica que utiliza bancos de dados geográficos como fonte de dados e implementa essa metodologia da G-GCS para representação dos fluxos locais. Com o uso de grafos cada vez maiores, o armazenamento e manipulação dessas informações passaram a ser um fator crucial, impondo assim uma grande restrição, limitando o processamento de grandes grafos. Com isso surge a necessidade de um modelo para armazenamento para grafos.

Os principais problemas encontrados foram:

- Alta ocupação da memória RAM;
- Limitação do tamanho dos dados a serem processados;
- Armazenamento incorreto dos grafos.

Assim, a motivação desta tese foi no sentido de criar um modelo para armazenamento e manipulação de grafos, o que implica também na definição de um modelo de *cache* para que grandes grafos possam ser manipulados.

1.2 Objetivos

Este trabalho tem como finalidade criar uma biblioteca para manipulação de grafos, dentro do contexto de geoprocessamento, criando um *framework* de simples utilização fornecendo um conjunto de classes que seja capaz de tratar diferentes tipos de grafos. Outro ponto que devemos destacar é a capacidade de tratamento de grandes

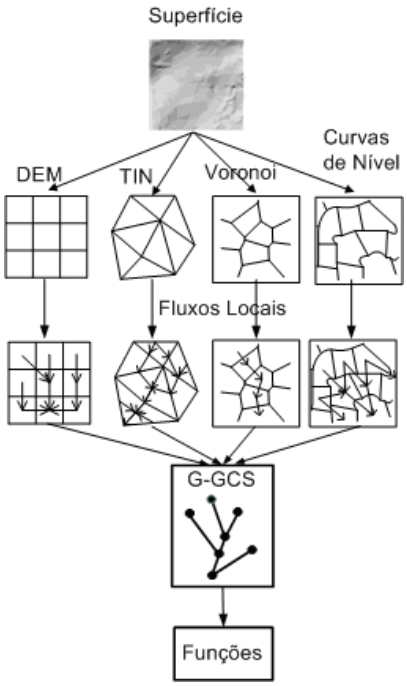


Figura 1.1 - Arquitetura da G-GCS.
Fonte: Rosim (2008).

quantidades de dados, independente de onde estejam armazenados, permitindo que diversas aplicações possam se beneficiar utilizando uma mesma biblioteca. Como os algoritmos de percorrimento serão fornecidos por bibliotecas de terceiros, é necessário uma boa definição dos elementos que representam o grafo para que esse "intercâmbio" de dados seja o mais simples possível.

Algumas informações referentes aos grafos são necessárias para que os mesmos possam ser armazenados e posteriormente manipulados. Os metadados referentes às estruturas do grafo são armazenados em bancos de dados relacionais. O armazenamento físico dos dados do grafo passa a ser flexível uma vez que os métodos de persistência são definidos como abstratos, ou seja, podem existir N implementações para acesso aos dados. Eles podem estar em um banco de dados, na memória ou descritos em arquivos utilizando linguagens específicas, tipo *The Graph Modelling Language - GML* (HIMSOLT, 1999), sendo necessário apenas que exista uma implementação concreta das funções de persistência em cada caso. Para este trabalho é escolhido o armazenamento dos dados em bancos de dados relacionais junto com seus metadados.

Os objetivos deste trabalho são:

- Objetivos Primários
  - Criar uma biblioteca para manipulação de grafos;
  - Dar suporte a diferentes tipos de grafos;
  - Manipular grandes quantidades de dados através de uma estrutura de *cache*.
- Objetivos Secundários
  - definição de classes de persistência e recuperação para permitir o acesso às informações em diferentes fontes de dados;
  - definição de estratégias de carregamento, definição de estratégias para a recuperação eficiente das informaçãos no banco;
  - criação de extratores, funções que são capazes de gerar grafos a partir de outras fontes de dados (vetoriais e matriciais).

1.3 Organização do Texto

Os capítulos restantes desta dissertação estão organizados da seguinte maneira:

- Fundamentação Teórica: apresenta resumidamente alguns tópicos abordados neste trabalho;
- Tecnologias Atuais: cita alguns projetos e aplicações correlacionados a este trabalho;
- Metodologia: descreve como o trabalho foi definido e implementado;
- Resultados: exemplifica os resultados obtidos;
- Conclusão: com base nos resultados obtidos são apresentadas as conclusões.

2 FUNDAMENTAÇÃO TEÓRICA

Para auxiliar no entendimento da metodologia adotada no desenvolvimento deste trabalho, é necessário uma breve explicação sobre alguns temas abordados e a definição de alguns conceitos.

2.1 Grafos

O artigo de *Leonhard Euler*, publicado em 1736, sobre o problema das sete pontes de *Königsberg* (Fig. 2.1), é considerado o primeiro resultado da teoria dos grafos (HARARY, 1969). O desafio era atravessar todas as pontes sem nenhuma repetição e *Euler* provou que não era possível.

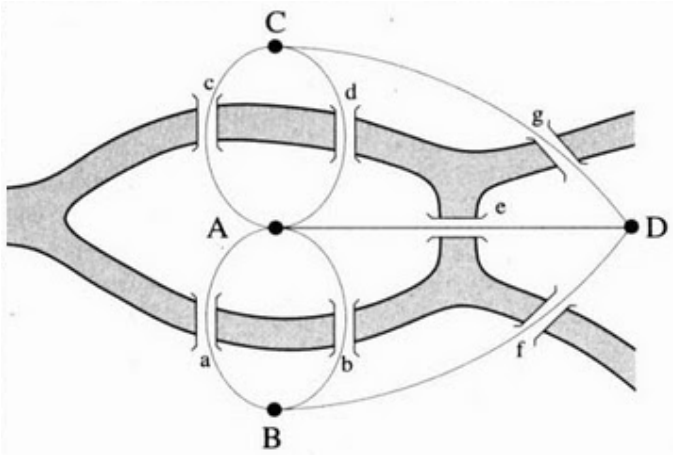


Figura 2.1 - Representação do problema das sete pontes de *Königsberg*.

2.1.1 Definição

Um grafo pode ser definido como sendo uma função $G = [N, A, \varphi]$, onde N é um conjunto de vértices, A é um conjunto de arestas e $\varphi(a)=(u,v)$ é uma função de incidência que associa cada aresta $a \in A$ a um par de vértices (u,v), $u \in N$ e $v \in N$ (BOLLOBÁS, 1998) (BONDY; MURTY, 2008).

Os grafos que possuem atributos associados aos seus vértices ou arestas são chamados de **redes** e auxiliam na modelagem de fenômenos, como fluxo de pessoas ou materiais, conexões de influência, linhas de comunicação e acessibilidade.

2.1.2 Conceitos Básicos

Seja um grafo simples (Fig. 2.2), representando um grafo com 6 vértices e 7 arestas.

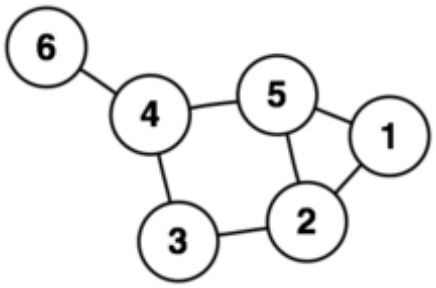


Figura 2.2 - Representação de um grafo simples.

$V = \{1,2,3,4,5,6\}$

$A = \{(1,2), (1,5), (2,3), (2,5), (3,4), (4,5), (4,6)\}$

Os conceitos associados à um grafo são:

- uma aresta representa a conexão de dois vértices e esses vértices são ditos como incidentes à aresta;
- o grau de um vértice é definido como o número de arestas incidentes sobre ele, sendo que as arestas internas (*loops*) são contadas duas vezes (BONDY; MURTY, 2008);
- dois vértices são considerados adjacentes se existe uma aresta entre eles;
- o conjunto de vizinhos de um vértice são todos os vértices adjacentes a ele;
- em um grafo orientado, difere-se o grau de saída (número de arestas saindo de um vértice) e grau de entrada (número de arestas chegando a um vértice).

Existem diversos casos específicos de grafos (Fig. 2.3), cada um com suas características. Este trabalho não irá tratar todos os tipos, mas irá dar suporte para que todos possam ser implementados através da definição de classes abstratas que representem o modelo genérico de um grafo.

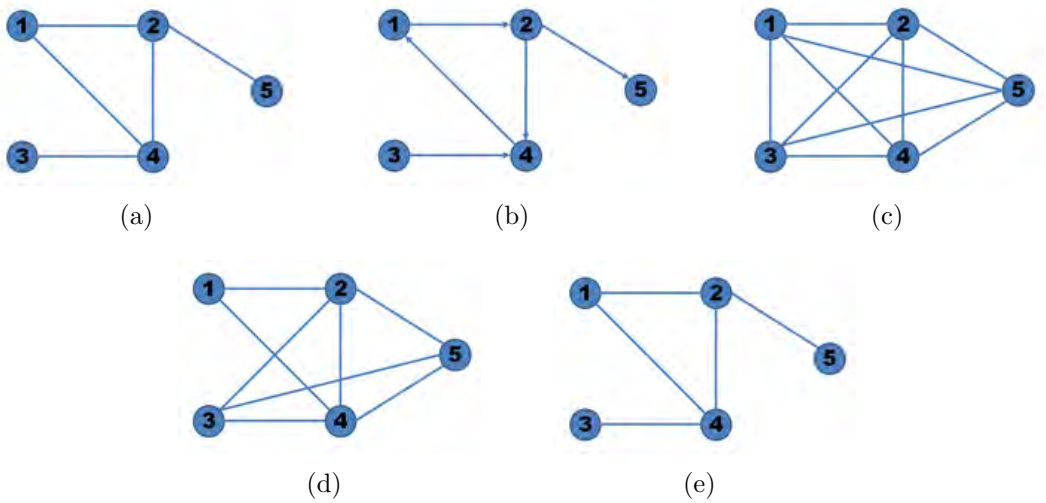


Figura 2.3 - Representação dos diversos tipos de grafos. (a) *Não Direcionado*; (b) *Direcionado*; (c) *Completo*; (d) *Denso*; (e) *Esparso*

2.1.3 Representação Computacional

Um grafo pode ser representado computacionalmente de diferentes formas, seja por listas ou em forma de matrizes. Em cada caso existem seus pontos positivos e negativos. Algumas dessas formas são:

- Lista de Adjacência

  Os vértices são armazenados como registros ou objetos, e cada vértice armazena uma lista de vértices adjacentes. Este tipo de estrutura de dados permite a adição de dados adicionais sobre os vértices.

- Lista de Incidência

  Os vértices e arestas são armazenadas como registros ou objetos. Cada vértice armazena suas arestas incidentes, e cada aresta armazena seus vértices incidentes. Esta estrutura de dados permite a adição de dados adicionais sobre vértices e arestas.

- Matriz de Adjacência

  Uma matriz de duas dimensões, em que as linhas representam os vértices de origem e as colunas representam os vértices de destino. Os dados sobre arestas e vértices devem ser armazenados em uma outra estrutura. Os valores presentes na matriz representam o custo de cada aresta entre um par de vértices.

- Matriz de Incidência

  Uma matriz bidimensional do tipo Booleana, em que as linhas representam os vértices e as colunas representam as arestas. As entradas indicam se o vértice de cada linha é incidente para a aresta de cada coluna.

As Listas de Adjacência são geralmente preferidas por sua eficiência na representação de grafos esparsos. Para o caso de grafos densos as Matrizes de Adjacência são preferidas, isto porque o número de arestas E é aproximadamente igual ao número de vértices ao quadrado V^2 ou em casos que é necessário procurar rapidamente se existe uma aresta ligando dois vértices dados (CORMEN et al., 2001).

Quando trabalha-se com bancos de dados, na maioria das vezes, a Matriz de Adjacência não é uma opção viável devido ao limite do número de colunas que uma tabela pode ter. Outro fator negativo são os casos em que ocorre a adição ou remoção de objetos, sendo necessário alterar a estrutura da tabela através da execução de SQL's (*Structured Query Language*) do tipo DDL (*Data Definition Language*). No caso de listas, apenas atualizações são feitas nas tabelas utilizando SQL's do tipo DML - (*Data Manipulation Language*). Neste trabalho o modelo de armazenamento adotado é por Lista de Incidência.

2.1.4 Operações em Grafos

Analisando as diversas bibliotecas de grafos é possível definir um conjunto comum de funções que são necessárias para se trabalhar com este tipo de estrutura de dados. Essas operações sobre um grafo G são:

- adjacent(G, x, y): verifica se existe uma aresta com os vértices x e y;
- neighbors(G, x): lista os vértices que possuem uma aresta partindo de x;
- add(G, x): adiciona um novo vértice ao grafo;
- add(G, x, y): adiciona uma nova aresta ao grafo;
- delete(G, x): remove o vértice x, se existir;
- delete(G, x, y): remove a aresta de vértices x e y, se existir;
- get_node_value(G, x): retorna o valor associado ao vértice x;
- set_node_value(G, x, a): define o valor associado ao vértice x para a.

Caso a estrutura dê suporte para associar valores às arestas, também são necessárias as seguintes funções:

- get_edge_value(G, x, y): retorna o valor associado à aresta *(x,y)*;
- set_edge_value(G, x, y, v): define o valor associado à aresta *(x,y)* para *v*.

2.2 Bancos de Dados Geográficos

É importante termos uma noção dos principais conceitos do que vêm a ser os bancos de dados geográficos pois utilizaremos algumas de suas características para auxiliar na recuperação dos dados dos grafos armazenados. Os bancos de dados geográficos (CASANOVA et al., 2005) são utilizados para o armazenamento e busca de informações relacionadas a objetos no espaço geográfico, incluindo pontos, linhas e polígonos.

Algumas de suas características são:

- oferece suporte aos tipos de dados espaciais em seu modelo de dados e em sua linguagem de pesquisa (SQL);
- suporta tipos de dados espaciais em sua implementação, o que permite indexações espaciais.

Além de todas as operações fornecidas pelos bancos de dados relacionais, algumas outras operações fornecidas por esse tipo de banco são:

- Mensuração: calcula a distância entre pontos e polígonos;
- Proximidade: busca objetos baseado na proximidade de um ponto específico;
- Região: buscam objetos contidos em uma região especificada.

Como a espacialidade é um atributo associado aos grafos nesse trabalho, é interessante explorarmos essas operações fornecidas por esses bancos para auxiliar na busca dos elementos armazenados.

Mas não bastam apenas os dados estarem presentes nesses bancos, é necessário uma correta indexação desses dados para que sua recuperação possa ser otimizada. Como

os vértices de um grafo normalmente são representados por pontos e espera-se que sua vizinhança esteja espacialmente próxima, uma boa estratégia de indexação seria utilizar a *R-tree*.

2.2.1 R-tree

R-trees (GUTTMAN, 1984) são estruturas de árvore de dados usadas em métodos de acesso espaciais para indexação multidimensional de informações, como coordenadas geográficas, retângulos ou polígonos. A ideia principal deste tipo de estrutura é agrupar objetos próximos e representá-los por um retângulo mínimo envolvente (MBR - *"'Minimum bounding rectangle"'*) a cada nível de sua árvore (Fig. 2.4).

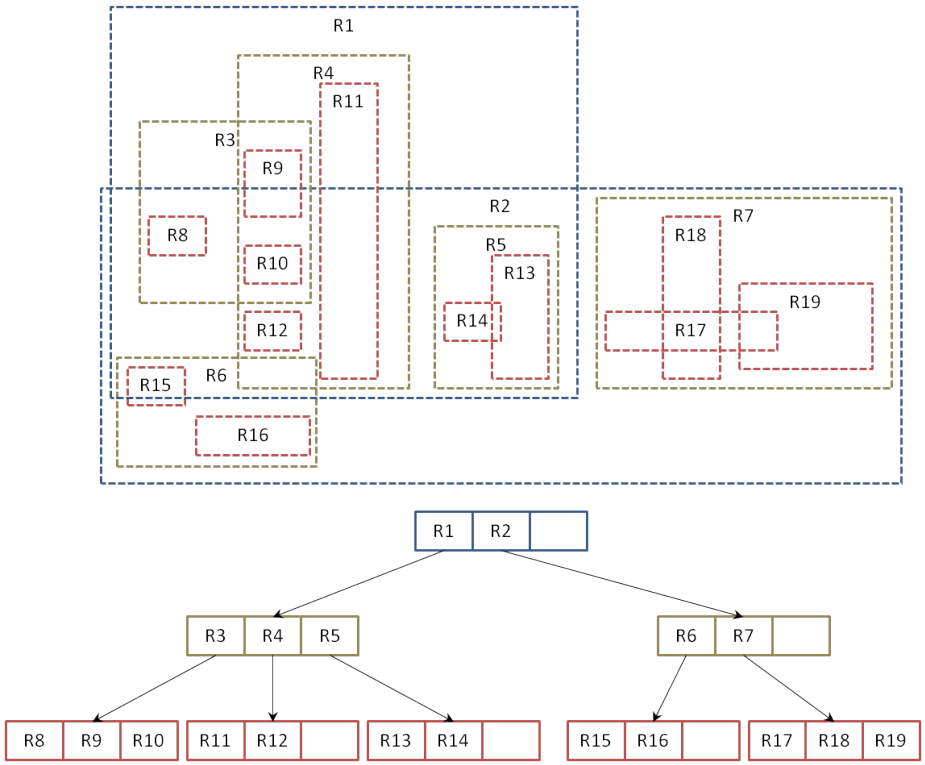


Figura 2.4 - Representação dos dados indexados por uma R-tree.

Com todos os objetos encontrando-se dentro deste retângulo envolvente, uma consulta que não intercepta o retângulo também não interceptará qualquer um dos objetos contidos. Isso é um fator muito importante na escolha desse tipo de indexação neste trabalho. Uma vez que cada objeto do grafo tenha associado a ele uma coordenada espacial, e esses dados estejam indexados, fica fácil a busca por um grupo de objetos. Na descrição das metodologias de acesso aos dados no banco de dados

(seção 4.4) esse conceito será mais detalhado.

2.3 Bancos de Dados orientados a Grafos

Neste trabalho são utilizados bancos de dados relacionais com extensão espacial para o armazenamento dos metadados dos grafos, como também é proposta uma forma de armazenar os grafos seguindo um esquema de tabelas. A extensão espacial nos auxilia na manipulação dos dados, mas isso não impede que sejam criadas outras implementações que façam o armazenamento dos grafos em outras fontes de dados. Por isso é importante ao menos citar a existência e algumas características dos bancos específicos para armazenamento de grafos.

Por definição, os bancos de dados especializados no armazenamento de grafos são os sistemas de armazenamento livres de indexação de adjacência, isso significa que todo elemento contém uma ligação direta com seus elementos adjacentes. Essa característica torna esses bancos bastante eficientes em operações como busca e percorrimento em grafos.

Um banco de dados orientado a grafos armazena as informações em uma rede de nós e arestas. As arestas representam o relacionamento entre os nós que representam os objetos (Fig. 2.5). Devido aos nós e arestas serem representados como objetos é possível associar conjuntos de atributos a eles.

Diversas implementações deste tipo de banco estão disponíveis. A seguir é apresentado um quadro com as informações a respeito de cada uma (Tab. 2.1).

2.4 *OGC - Open Geospatial Consortium*

O *Open Geospatial Consortium* (OGC, 2011) é um consórcio internacional sem fins lucrativos, formado por instituições voluntárias cuja finalidade é desenvolver padrões que facilitem o acesso e a troca de informações espaciais. Dentre os diversos padrões desenvolvidos pelo OGC, estão os padrões de representação e armazenamento de feições vetoriais. O padrão *Simple Features Specification (SFS)* (RYDEN, 2005) define um modelo para o armazenamento de feições vetoriais simples. Este modelo de feições vetoriais se baseia em uma hierarquia de classes (Fig. 2.6).

Não tem definido no padrão SFS nenhuma feição que represente especificamente o conceito de rede ou mesmo de grafo. Em (RYDEN, 2005) cita que é possível utilizar uma *Multilinestring*, conjunto de linhas, para a representação de redes.

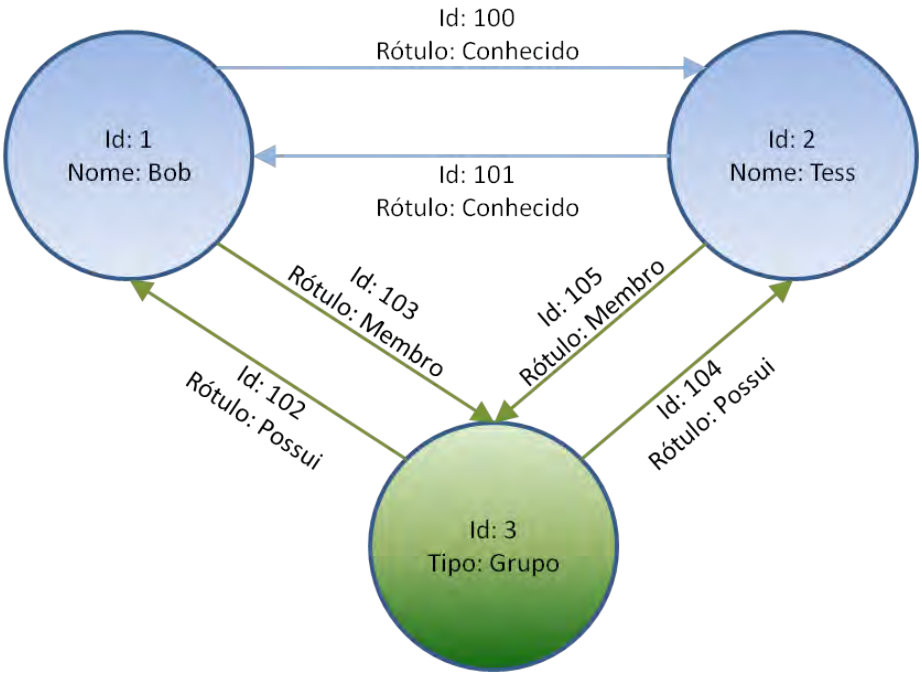


Figura 2.5 - Objetos representando os vértices e arestas no banco de dados.

Tabela 2.1 - Implementações de Bancos de Dados orientados a Grafos.

Implementação	Linguagem	Modelo	Persistência	Licença
Neo4j	Java	Próprio	Próprio em Disco	AGPLv3
OrientDB	Java 6	Próprio (esquema de vértices e arestas)	Próprio em Disco	Apache 2.0
DEX	Java, C++	Multi grafo rotulado e direcional	Disco	Livre com restrição
InfoGrid	Java	Grafo OO, suporta modelo de semântica	-	AGPLv3
HyperGraphDb	Java	Grafo OO, multi relacional e rotulado	-	LGPL
Infinitegraph	Java, C++	Multi grafo rotulado e direcional	via Objectivity/DB	Código fechado
sones	C#	Próprio e OO	Próprio em Disco	AGPLv3

Fonte: graph-database.org (2011).

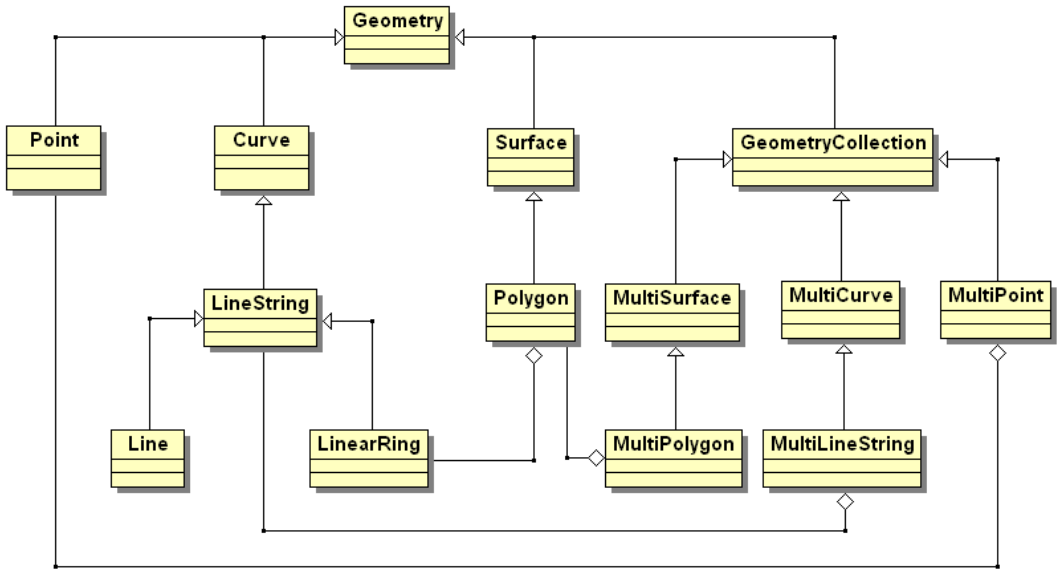


Figura 2.6 - Hierarquia das estruturas vetoriais proposta pelo modelo SFS.
Fonte: Adaptado de OGC (2011).

2.5 TerraLib 5

Uma definição geral de *Middleware* pode ser entendida como sendo uma camada de *software* que não representa a aplicação final e auxilia na interoperabilidade dos dados, criando um desacoplamento entre aplicação e fontes de dados. Seguindo esse conceito é que iremos utilizar a TerraLib para a criação de um *framework* para grafos, possibilitando que aplicações distintas possam utilizar o mesmo arcabouço e fiquem independentes de como os dados serão persistidos.

A TerraLib é definida como um projeto (CAMARA et al., 2008) que visa atender grandes demandas institucionais na área de Geoinformática, criando um ambiente para pesquisa e desenvolvimento de inovações em geoprocessamento.

A TerraLib 5 (Fig. 2.7), é uma plataforma de *software* tendo seu núcleo escrito em C++ e tem como principais características (QUEIROZ et al., 2010):

- Acesso aos dados: acesso a diferentes tipos de fontes de dados (SGBD's, dados vetoriais, imagens, serviços web entre outros);
- Persistência/mapeamento: mapeia dados de diversas fontes de dados para diferentes finalidades;
- Estruturas de agregação: fornece um mecanismo extensível capaz de intro-

duzir novas representações de alto nível.

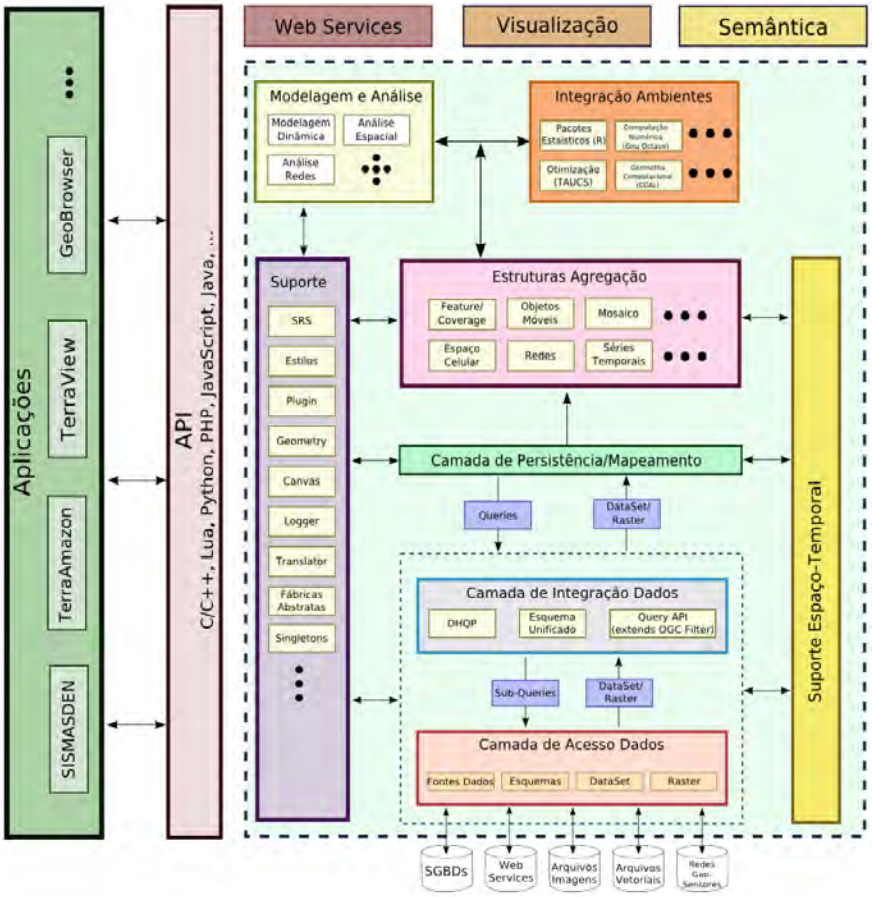


Figura 2.7 - Esquema de representação da Terralib5.
Fonte: Queiroz et al. (2010).

Baseado nessa última característica, que nos permite criar novos tipos de representação, e utilizando das facilidades de acesso aos dados é que iremos implementar nosso projeto fazendo uso dessa tecnologia. Podemos ver o modelo de grafo proposto neste trabalho como sendo uma estrutura de agregação da TerraLib.

3 TECNOLOGIAS ATUAIS

3.1 Algoritmos em SQL

Uma possível abordagem para se tentar resolver os problemas de armazenamento e processamento dos dados de um grafo é armazená-los em bancos de dados relacionais. Utilizando a flexibilidade fornecida pela linguagem de consulta SQL dos bancos pode-se realizar algumas operações.

No exemplo a seguir, duas tabelas são criadas para representar os vértices e arestas (ALBERTON, 2010).

```
CREATE TABLE nodes (
 id INTEGER PRIMARY KEY,
 name VARCHAR(10) NOT NULL,
 feat1 CHAR(1),
 feat2 CHAR(1)
);

CREATE TABLE edges (
 a INTEGER NOT NULL REFERENCES nodes(id)
   ON UPDATE CASCADE ON DELETE CASCADE,
 b INTEGER NOT NULL REFERENCES nodes(id)
   ON UPDATE CASCADE ON DELETE CASCADE,
 PRIMARY KEY (a, b)
);

CREATE INDEX a_idx ON edges (a);
CREATE INDEX b_idx ON edges (b);
```

Uma consulta simples para descobrir quais vértices estão conectados ao vértice "1" pode ser definido como:

```
SELECT *
  FROM nodes n
  LEFT JOIN edges e ON n.id = e.b
 WHERE e.a = 1;  -- retrieve nodes connected to node 1
```

Através da utilização de comandos especiais dos bancos de dados e com a utilização de tabelas temporárias é possível definir métodos de busca, por exemplo o método BFS (*Breadth First Search*). No exemplo a seguir é assumido uma tabela de arestas (parent_id int, child_id int), partindo de uma raiz indicada por (@root_id), a função irá retornar todos os vértices alcançáveis.

```
create table \#reached( id int primary key )
insert into \#reached values ( @root\_id )
while ( @@rowcount > 0 ) begin
  insert into \#reached (id ) select distinct child\_id
  from edges e join \#reached p on p.id = e.parent\_id
  where e.child_id not in ( select id from \#reached )
end
```

Para esses casos de consultas complexas existe a necessidade de se criar funções (*Stored Procedures*) que devem ser armazenadas no próprio servidor de banco de dados e são definidas utilizando-se uma linguagem específica chamada PLSQL (*Procedural Language/Structured Query Language*), ou fazendo-se uso de funções que são particulares a um específico SGBD.

Normalmente as funções que são definidas para um banco não se aplicam aos outros devido a pequenas diferenças de sintaxe, o que dificulta esse tipo de abordagem neste trabalho. Outro fator negativo é a necessidade dessas funções terem que estar presentes no servidor. Apenas as *SQL's* que são aplicáveis a todos SGBD's serão utilizadas neste trabalho.

3.2 Bancos de Dados Geográficos

Em relação aos SGBDs com suporte a dados espaciais e que também tratam o armazenamento de grafos é interessante observarmos seus modelos de armazenamento pois se assemelham com o que está proposto neste trabalho.

3.2.1 Oracle Network

O *Oracle Network* (MURRAY, 2009) é uma plataforma proprietária utilizada para a modelagem e análise em aplicações de redes. Apresenta dois modelos de dados, um utilizando a informação de topologia para seu armazenamento e a outra utilizando a informação de conexão da rede.

Esta plataforma define o conceito de redes como sendo um conjunto de nós e *links*. Cada *link* (também chamado de aresta ou segmento) define dois nós. A rede pode ser direta (um nó inicial e final definem a direção do *link*) ou indireta (os *links* podem ser atravessados por ambas as direções).

Alguns conceitos definidos são:

- um nó representa um objeto de interesse e pode ter um par de coordenadas associados que descreve sua localização espacial;
- um *link* representa a relação entre dois nós, podendo ser direcionais ou não;
- um caminho é uma sequência alternada de nós e *links*, começando e terminando com um nó. Normalmente os nós e *links* aparecem apenas uma vez em um caminho.

Uma característica interessante deste modelo é a possibilidade de otimizar o uso de memória. O conceito *Load on Demand* determina uma forma de particionar a rede, auxiliando no momento em que operações de análises sejam feitas. Uma partição de uma rede não será carregada para memória enquanto a partição carregada não terminar de realizar toda a sua análise.

Para a execução dessa estratégia é necessário um pré-processamento dos dados antes que eles possam ser utilizados de fato. Isso é feito seguindo-se os seguintes passos:

Criação da Rede → Particionamento da Rede → Configuração do *Cache* → Análise da Rede.

As tabelas que descrevem seu modelo de armazenamento são apresentadas abaixo.

- Tabela de Nós:

  Analizando a Tabela 3.1 utilizada para o armazenamento de nós vemos atributos específicos para representar a geometria, o custo e para a definição da hierarquia do nó. O modelo de armazenamento dos dados proposto neste trabalho nos permite a associação de diversos atributos aos nós, fazendo com que possamos ter a mesma capacidade de representação deste modelo do *Oracle Network*.

- Tabela de *Links*:

  Da mesma forma que ocorre na tabela de nós, na tabela de *links* (Tab. 3.2) existem atributos para representar a geometria e o custo.

- Tabela de Caminhos:

  A definição de caminho (Tab. 3.3) é uma abordagem que não foi feita neste trabalho.

Tabela 3.1 - Colunas da tabela de Nós do modelo *Oracle Network*.

Nome	Tipo	Descrição
NODE_ID	NUMBER	Identificação do nó
NODE_NAME	VARCHAR(32)	Nome do nó
NODE_TYPE	VARCHAR(24)	Tipo do nó definido pelo usuário
ACTIVE	VARCHAR(1)	Contém Y se o nó está visível na rede ou N em caso contrário
GEOM_ID	SDO_GEOMETRY	Para redes espaciais, SDO_GEOMETRY associado a este nó
cost_column	NUMBER	Valor de custo associado a este nó
HIERARCHY_LEVEL	NUMBER	Para redes hierárquicas
PARENT_NODE_ID	NUMBER	Para redes hierárquicas

Fonte: Murray (2009).

Tabela 3.2 - Colunas da tabela de *Links* do modelo *Oracle Network*.

Nome	Tipo	Descrição
LINK_ID	NUMBER	Identificação do *link*
LINK_NAME	VARCHAR(32)	Nome do *link*
START_NODE_ID	NUMBER	Identificação do nó de origem deste *link*
END_NODE_ID	NUMBER	Identificação do nó de destino deste *link*
LINK_TYPE	VARCHAR(24)	Tipo do *link* definido pelo usuário
ACTIVE	VARCHAR(1)	Contém Y se o *link* está visível na rede ou N em caso contrário
GEOM_ID	SDO_GEOMETRY	Para redes espaciais, SDO_GEOMETRY associado a este *link*
cost_column	NUMBER	Valor de custo associado a este *link*
PARENT_LINK_ID	NUMBER	Para redes hierárquicas
BIDIRECTED	VARCHAR(1)	Para redes direcionais, contém Y se for não-direcional e N se for direcional

Fonte: Murray (2009).

- Tabela de Metadados:

A tabela de metadados (Tab. 3.4) é formada por um conjunto de atributos que descrevem a rede armazenada.

Tabela 3.3 - Colunas da tabela de Caminho do modelo *Oracle Network*.

Nome	Tipo	Descrição
PATH_ID	NUMBER	Identificação do caminho
PATH_NAME	VARCHAR(32)	Nome do caminho
PATH_TYPE	VARCHAR(24)	Tipo do caminho definido pelo usuário
START_-NODE_ID	NUMBER	Identificação do nó de origem do primeiro *link* deste caminho
END_NODE_ID	NUMBER	Identificação do nó de destino do último *link* deste caminho
COST	NUMBER	Valor de custo associado ao caminho
SIMPLE	VARCHAR(1)	Contém Y se o caminho é simples (único) ou N se for complexo
geom_column	SDO_GEOME-TRY	Para redes espaciais, SDO_GEOMETRY é associado a este caminho

Fonte: Murray (2009).

O modelo adotado pelo *Oracle Network* permite o tratamento tanto de redes direcionais como hierárquicas através da definição de atributos específicos em seu modelo de tabelas. O modelo deste trabalho terá algumas dessas informações separadas em duas tabelas, uma sendo de metadados do grafo (tipo de grafo, modo de armazenamento, etc.) e outra referente aos atributos associados aos vértices e arestas (geometria, custo, etc.).

3.2.2 PgRouting

Uma extensão de bancos de dados geográficos que também aborda o armazenamento de grafos é o *pgRouting*, com o detalhe de ser uma ferramenta não proprietária.

O *pgRouting* (KASTL; JUNOD, 2010) é uma extensão do *PostGIS* que acrescenta funcionalidades de roteamento ao *PostGIS/PostgreSQL*. Algumas das vantagens em se utilizar esse banco de dados para roteamento são:

- acessível através de vários clientes ou diretamente por PL/SQL;
- uso do *PostGIS* para suporte aos dados geográficos;
- SIG's podem acessar e modificar seus dados e atributos.

A seguir iremos demonstrar a utilização do *pgRouting* com o intuito de mostrar como alguns passos são frequentes em todas as aplicações. Interessante notar que as

Tabela 3.4 - Principais colunas da tabela de Metadados do modelo *Oracle Network*.

Nome	Tipo	Descrição
OWNER	VARCHAR(32)	Dono da rede
NETWORK	VARCHAR(24)	Nome da rede
NETWORK_ID	NUMBER	Identificação da rede
NETWORK_-CATEGORY	VARCHAR(12)	*SPATIAL* ou *LOGICAL*
GEOMETRY_-TYPE	VARCHAR(24)	Contém o tipo da geometria dos nós e *links*
NETWORK_-TYPE	VARCHAR(24)	Tipo da rede definida pelo usuário
NODE_TA-BLE_NAME	VARCHAR(32)	Nome da tabela que possui as geometrias associadas aos nós
NODE_-GEOM_CO-LUMN	VARCHAR(32)	Nome da coluna que possui as informações de geometria dos nós
LINK_TABLE_-NAME	VARCHAR(32)	Nome da tabela que possui as geometrias associadas aos *links*
LINK_GEOM_-COLUMN	VARCHAR(32)	Nome da coluna que possui as informações de geometria dos *links*
LINK_DIREC-TION	VARCHAR(12)	*DIRECTED* ou *INDIRECTED*
PATH_TA-BLE_NAME	VARCHAR(32)	Nome da tabela que possui as geometrias associadas aos caminhos
PATH_GEOM_-COLUMN	VARCHAR(32)	Nome da coluna que possui as informações de geometria dos caminhos

Fonte: Murray (2009).

tabelas devem ser ajustadas para que as funções tenham o comportamento esperado, e que cada função é definida por um arquivo SQL. O exemplo utilizado foi extraído de (KASTL; JUNOD, 2010).

Criando um banco e carregando um arquivo de dados:

```
createdb -U postgres pgrouting-test
psql -U postgres -d pgrouting-test \ -f ~/data/sampledata.sql
```

Adicionando as funções de roteamento ao servidor:

```
psql -d routing -f
 /usr/share/postgresql/8.4/contrib/postgis-1.5/postgis.sql
psql -d routing -f
 /usr/share/postgresql/8.4/contrib/postgis-1.5/spatial_ref_sys.sql
```

```
psql -d routing -f /usr/share/postlbs/routing_core.sql
psql -d routing -f /usr/share/postlbs/routing_core_wrappers.sql
psql -d routing -f /usr/share/postlbs/routing_topology.sql
```

As tabelas importadas do arquivo sampledata.sql são as tabelas comuns em qualquer banco espacial além da tabela *ways* que representa os dados (Tab. 3.5):

Tabela 3.5 - Tabelas importadas, exemplo de utilização do *pgRouting*.

Esquema	Nome	Typo	Dono
público	geography_columns	view	postgres
público	geometry_columns	table	postgres
público	spatial_ref_sys	table	postgres
público	ways	table	postgres

A tabela *ways* representa os dados de uma rodovia, possuindo os seguintes atributos (Tab. 3.6):

Tabela 3.6 - Tabela *ways*, exemplo de utilização do *pgRouting*.

Coluna	Tipo	Modificadores
gid	integer	not null
class_id	integer	table
length	double	table
name	character	table
the_geom	geometry	table

Seus índices são:

- "ways_pkey" - chave primária, coluna *ways*,
- "geom_dx" - indexação espacial (btree), coluna *the_geom*.

Para o cálculo da topologia é necessário que os dados tenham as informações sobre origem e destino de cada rodovia para que seja possível executar a função de criação de topologia.

```
ALTER TABLE ways ADD COLUMN ‘‘source’’ integer;
ALTER TABLE ways ADD COLUMN ‘‘target’’ integer;
```

```
SELECT assign_vertex_id('ways', 0.00001, 'the_geom', 'gid');
```

Em casos de grande quantidade de dados é necessário que se crie uma indexação para auxiliar na recuperação das informações.

```
CREATE INDEX source_idx ON ways(``source'');
CREATE INDEX target_idx ON ways(``target'');
```

O algoritmo de Dijkstra (problema do caminho mais curto) é executado através da seguinte chamada de função:

```
shortest_path( sql text,
        source_id integer,
        target_id integer,
        directed boolean,
        has_reverse_cost boolean )
```

Note que *shortest_path* é uma função interna do *pgRouting*, pertencendo apenas a este tipo de banco de dados, igual como foi mostrado na sessão anterior.

O que vemos nesse exemplo são algumas similaridades com o que estamos propondo, tais como: necessidade de se utilizar uma indexação espacial para auxiliar na recuperação das informações e definição de um modelo de tabelas que represente explicitamente a ligação entre os elementos do grafo.

3.3 Bancos de Dados orientados a Grafos

Mesmo que os bancos de dados orientados a grafos não sejam o foco desse trabalho, em função do interesse na componente espacial dos dados que são tratadas pelos bancos de dados relacionais com extensão espacial, é interessante observarmos o funcionamento de sua API e com que tipos de objetos eles trabalham.

3.3.1 Neo4j

Neo4j é um dos mais conhecidos bancos de dados orientados a grafos, sendo otimizado para estruturas de grafos ao invés de tabelas (TEAM, 2012). Pode ser definido como (Fig. 3.1):

A Figura 3.1 pode ser entendida do seguinte modo:

Um Grafo - registra seus dados em → Nós - que possuem → Propriedades

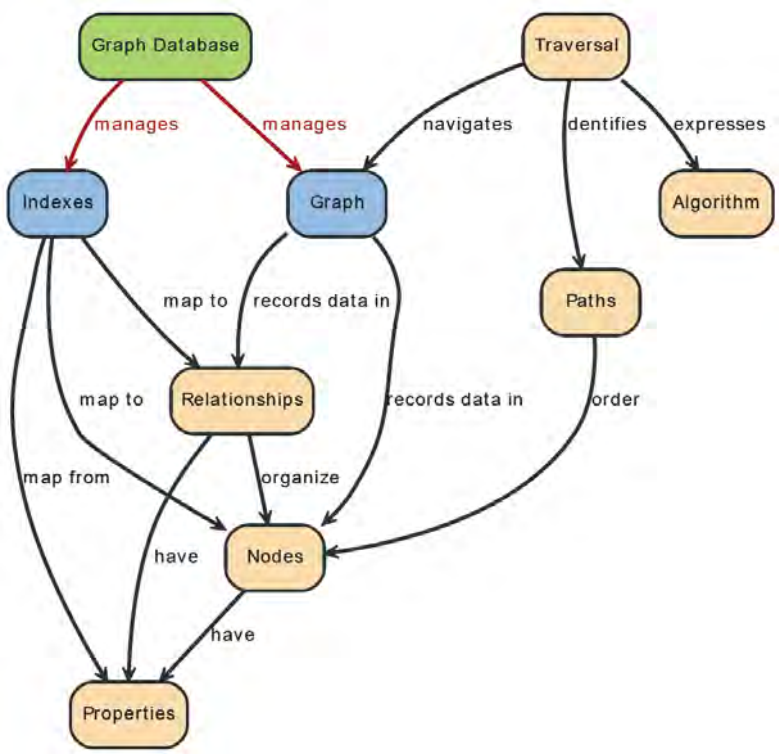


Figura 3.1 - Definição do modelo utilizado em Neo4j.
Fonte: Team (2012).

Nós - são organizados por → Relações - que também possuem → Propriedades

Travessia - navega → um Grafo - identifica → Caminhos - que ordena → Nós

Índices - mapeiam → Propriedades - para → Nós ou Relações

Graph Database (Neo4j) - controla um → Grafo - e também controla → Índices

A seguir é apresentado um *Hello World* demonstrando a utilização de sua *API*, adaptado de Team (2012).

Inicializando algumas variáveis:

```
GraphDatabaseService graphDb;
Node firstNode;
Node secondNode;
Relationship relationship;
```

Inicializando o servidor de banco de dados:

```
graphDb = new GraphDatabaseFactory().newEmbeddedDatabase( DB_PATH );
```

```
registerShutdownHook( graphDb );
```

Criando um pequeno grafo:

```
firstNode = graphDb.createNode();
firstNode.setProperty( "message", "Hello, " );
secondNode = graphDb.createNode();
secondNode.setProperty( "message", "World!" );
relationship = firstNode.createRelationshipTo( secondNode,
RelTypes.KNOWS );
relationship.setProperty( "message", "brave Neo4j " );
```

Imprimindo o resultado:

```
System.out.print( firstNode.getProperty( "message" ) );
System.out.print( relationship.getProperty( "message" ) );
System.out.print( secondNode.getProperty( "message" ) );
```

O resultado obtido é:

```
Hello, brave Neo4j World!
```

Percebemos que os bancos de dados deste modelo estão sempre associados a um caminho que define um arquivo em disco, utilizado para a persistência dos dados. Outro detalhe é em relação à associação de atributos aos objetos, que é feito de maneira simples sem a preocupação de esteriotipação (tipo do dado).

3.3.2 DEX

DEX pode ser definido (TECHNOLOGIES, 2011b) como sendo um sistema gerenciador de banco de dados orientado a multigrafos, rotulado e direcionado. Rotulado porque todos os nós e arestas podem ter atributos para classificá-los, direcionado porque suporta arestas com direção e também multigrafo porque não possui restrição sobre o número de arestas entre dois nós. É importante ressaltar que *DEX* utiliza banco de dados embarcado.

Como *DEX* possui uma API em C++, será mostrado em mais detalhes um exemplo de sua utilização para que se possa entender suas capacidades, códigos adaptados de (TECHNOLOGIES, 2011a). Considere o grafo (Fig. 3.2):

Criando um banco de dados e uma sessão:

```
DexConfig cfg;
```

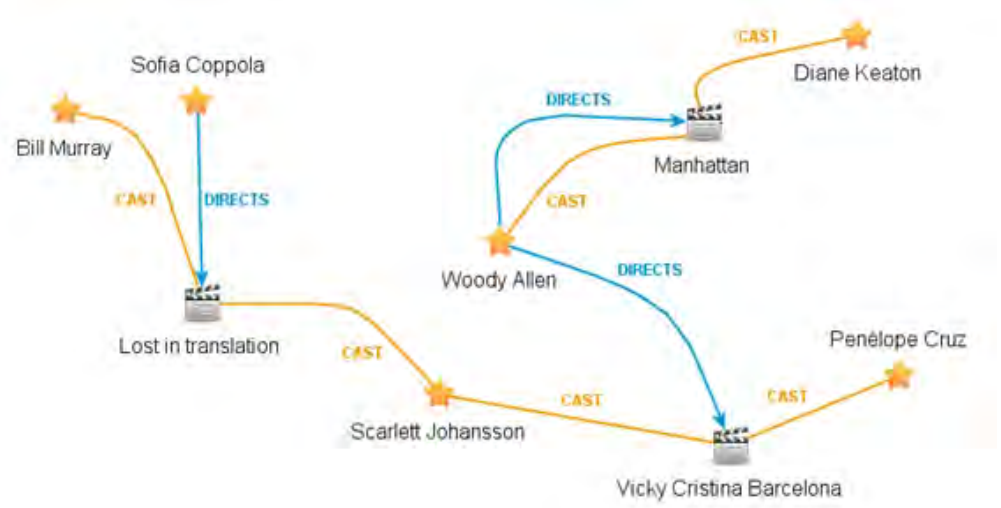


Figura 3.2 - Exemplo de um grafo.
Fonte: Technologies (2011b).

```
cfg.SetLicense(``Your license key'');
Dex *dex = new Dex(cfg);
Database *db = dex->Create(``HelloDex.dex'', ``HelloDex'');

Session *sess = db->NewSession();
Graph *g = sess->GetGraph();
```

Adicionando os Nós:

```
Value *value = new Value();
//MOVIES
oid_t mLostInTranslation = g->NewNode(movieType);
g->SetAttribute(mLostInTranslation, movieTitleType,
        value->SetString(``Lost in Translation''));

oid_t mVickyCB = g->NewNode(movieType);
g->SetAttribute(mVickyCB, movieTitleType,
        value->SetString(``Vicky Cristina Barcelona''));
...
//PEOPLES
oid_t pScarlett = g->NewNode(peopleType);
g->SetAttribute(pScarlett, peopleNameType,
        value->SetString(``Scarlett Johansson''));
oid_t pBill = g->NewNode(peopleType);
g->SetAttribute(pBill, peopleNameType,
        value->SetString(``Bill Murray''));
```

```
...
```

Adicionando as arestas:

```
oid_t anEdge;
anEdge = g->NewEdge(castType, mLostInTranslation, pScarlett);
g->SetAttribute(anEdge, castCharacterType,
        value->SetString(``Charlotte''));

anEdge = g->NewEdge(castType, mLostInTranslation, pBill);
g->SetAttribute(anEdge, castCharacterType,
        value->SetString(``Bob Harris''));
...
```

Exemplos de pesquisas:

```
Objects *castDirectedByWoody =
        g->Neighbors(directedByWoody, castType, Any);
```

Iteradores:

```
ObjectsIterator *it = castDirectedByWoody->Iterator();
while (it->HasNext())
{
        oid_t peopleOid = it->Next();
        g->GetAttribute(peopleOid, peopleNameType, *value);
        std::wcout << "`Hello "` << value->GetString() << std::endl;
}
delete it;
delete castDirectedByWoody;
```

Fechando o banco de dados:

```
delete sess;
delete db;
delete dex;
```

Neste modelo, como no anterior, também é necessário a definição de um arquivo em disco para o armazenamento do grafo. Diferentemente do modelo anterior, a associação de atributos aos objetos depende do tipo do dado, função "*setString*". Este modelo suporta a utilização de iteradores, que são utilizados em casos que seja necessário acessar todos os elementos do grafo independente da forma que seus elementos estão conectados. Outra característica é que cada banco de dados possui

um grafo associado, permitindo que o banco seja manipulado como sendo um único grafo. Algumas das características desse banco são:

- o grafo pertence a um banco de dados e a uma sessão;
- nós e arestas podem ter atributos associados;
- não existe limite de arestas entre dois nós;
- nós e arestas possuem um identificador único no grafo.

Todos os dados de um banco de dados são armazenados em um arquivo que somente pode ser aberto através da API do *DEX*. A manipulação de seus dados somente pode ocorrer dentro de uma sessão. Múltiplos bancos de dados não compartilham a memória, ou seja, não existe comunicação entre eles.

3.4 Aplicações TerraLib

Mesmo dentro do ambiente TerraLib, é possível encontrar diversas implementações de grafos. Diferentes aplicações foram construidas, mas nunca com a preocupação de se definir um modelo único de representação para os grafos.

Algumas dessas aplicações são:

- TerraHidro (ROSIM et al., 2003): Aplicação para modelagem hidrológica.
- Flow: Plugin do TerraView de fluxos baseado no trabalho (OLIVEIRA, 2005).
- Re-Seg (KORTING, 2007): Plugin do TerraView para re-segmentação de imagens.

Cada uma dessas aplicações trata problemas distintos, mas poderiam utilizar o mesmo arcabouço no seu desenvolvimento por manipularem os mesmos tipos de dados, os grafos.

A seguir é apresentada uma tabela (Tab. 3.7) comparativa, levando-se em conta algumas das características abordadas neste trabalho:

Tabela 3.7 - Tabela comparativa das aplicações TerraLib que tratam grafos.

	Graph	Hidro	Flow	Re-Seg
Extensível	X	X	-	-
Cache	X	-	-	-
Persistência	X	-	-	-
Manipulação	X	X	X	X
Iterador	X	-	-	-
Visualizador	X	X	X	-

3.5 Conclusões

Neste capítulo foram apresentadas diversas soluções para o tratamento de grafos, sejam bancos de dados ou bibliotecas computacionais, sempre encontramos alguns pontos em comuns e outros que tornam cada opção única.

Para este trabalho foram utilizadas algumas características e conceitos encontrados nas aplicações descritas, são elas:

- utilização de *SQL's* que sejam aplicáveis a todos os bancos de dados relacionais para evitar dependências;
- definição de uma tabela de metadados, devido a necessidade de existir uma tabela que descreva e identifique de forma correta todos os grafos presentes no banco de dados;
- definição de uma API que permita o acesso e manipulação dos dados de forma simples, independente de como e onde os dados estejam armazenados.

Os detalhes deste *framework* serão abordados no próximo capítulo.

4 METODOLOGIA

Neste capítulo será apresentada a metodologia utilizada no desenvolvimento deste trabalho, bem como os detalhes de cada estratégia adotada.

4.1 Modelo de Persistência

Como dito anteriormente, para que um grafo possa ser armazenado em um SGBD e posteriormente recuperado é necessário um conjunto de metadados que o descreva. Informações do tipo:

- quais as características do grafo;
- quais atributos estão sendo associados ao grafo;
- onde e como os dados do grafo estão armazenados.

são de fundamental importância para sua manipulação. A Figura 4.1 exemplifica o modelo adotado para armazenamento dos grafos em bancos de dados.

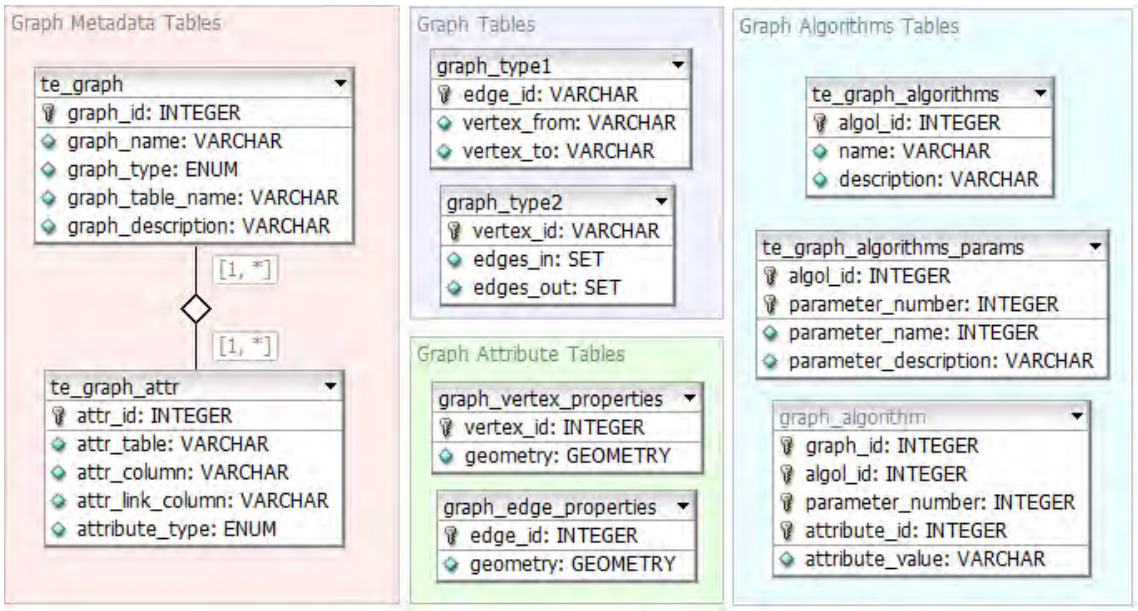


Figura 4.1 - Modelo de tabelas proposto para armazenamento de grafos.

As tabelas estão dividas nos seguintes blocos:

- *Graph Metadata Tables*: são as tabelas utilizadas para identificação dos

grafos e os atributos pertencentes a cada um;

- *Graph Tables*: são as tabelas utilizadas para armazenamento dos dados dos grafo, é proposto um modelo de armazenamento por arestas, mas isso não implica que este seja único;
- *Graph Attribute Tables*: tabelas utilizadas para associar atributos aos grafos, isso não impede que os atributos estejam diretamente nas tabelas de dados, mas sim permitir a associação de atributos de outras tabelas aos grafos;
- *Graph Algorithms Tables*: são as tabelas destinadas para a descrição dos algoritmos disponíveis para processamento.

Apenas as tabelas com prefixo "*te_*" fazem parte deste modelo conceitual, as demais tabelas serão utilizadas conforme necessidade do usuário. As tabelas "graph_type1" e "graph_type2" são exemplos de tabelas utilizadas para o armazenamento dos dados do grafo. Ambas as tabelas possuem a semântica do grafo, mas armazenadas de modo diferente, ficando a critério do usuário escolher qual modelo utilizar. A princípio este trabalho irá dar suporte para o armazenamento utilizando listas de arestas, modelo da tabela "graph_type1".

Este modelo de tabelas permite associar N atributos a um grafo, inclusive associar um mesmo atributo a grafos diferentes sem a necessidade de duplicar essas informações. As tabelas "graph_vertex_properties" e "graph_edge_properties" são exemplos de tabelas do banco que podem conter atributos do grafo, inclusive o atributo com sua geometria espacial que o represente.

A seguir é apresentado uma descrição detalhada de cada tabela do modelo.

4.1.1 Tabela te_graph

A tabela *te_graph* é responsável por armazenar as informações que descrevem o objeto *grafo*, ou seja, seus metadados. A Tabela 4.1 apresenta seus campos.

- graph_id: identificador único do grafo;
- graph_name: nome associado ao grafo;
- graph_type: enumeral que define o tipo do grafo;
- graph_table_name: nome da tabela com os dados do grafo;

Tabela 4.1 - Tabela *te_graph*.

Nome	Tipo	Não é NULL	Chave
graph_id	INTEGER	Sim	Sim
graph_name	VARCHAR	Não	Não
graph_type	INTEGER	Não	Não
graph_table_-name	VARCHAR	Não	Não
graph_descrip-tion	VARCHAR	Não	Não

- graph_description: descrição sobre o grafo.

Os metadados presentes nesta tabela são suficientes para que um grafo possa ser localizado no banco de dados, mas não existe a necessidade desta restrição. Esta tabela abre uma série de discussões a respeito de quais seriam as informações necessárias para se localizar um grafo, não importando sua localização, seja em disco, em algum servidor ou mesmo nas nuvens.

Os comandos *SQL* necessários para a construção desta tabela são apresentados abaixo.

```
CREATE TABLE te_graph (
  graph_id serial NOT NULL,
  graph_name VARCHAR NULL,
  graph_type INTEGER NULL,
  graph_table_name VARCHAR NULL,
  graph_description VARCHAR NULL,
  PRIMARY KEY(graph_id)
);
```

4.1.2 Tabela te_graph_attr

A tabela *te_graph_attr* é responsável por armazenar as informações dos atributos que estão associados aos grafos. A Tabela 4.2 mostra seus campos.

- attr_id: identificador único do atributo;
- attr_table: nome da tabela que possui este atributo;
- attr_column: nome da coluna que identifica o atributo;
- attr_link_column: nome da coluna de ligação para associação ao objeto;

Tabela 4.2 - Tabela *te_graph_attr*.

Nome	Tipo	Não é NULL	Chave
attr_id	INTEGER	Sim	Sim
attr_table	VARCHAR	Não	Não
attr_column	VARCHAR	Não	Não
attr_link_column	VARCHAR	Não	Não
attribute_type	INTEGER	Não	Não

- attribute_type: enumeral que descreve o tipo do atributo.

Os comandos *SQL* necessários para a construção desta tabela é apresentado abaixo.

```
CREATE TABLE te_graph_attr (
  attr_id serial NOT NULL,
  attr_table VARCHAR NULL,
  attr_column VARCHAR NULL,
  attr_link_column VARCHAR NULL,
  attribute_type INTEGER NULL,
  PRIMARY KEY(attr_id)
);
```

4.1.3 Tabela te_graph_algorithms

A tabela *te_graph_algorithms* é utilizada para indicar quais os algoritmos estão disponíveis para serem executados e a tabela *te_graph_algorithms_params* indica quais os parâmetros utilizados em cada um dos algoritmos para sua execução.

Se um grafo foi gerado através de um algoritmo, as informações de geração são representadas na tabela "graph_algorithm", indicando qual algoritmo e parâmetros foram utilizados.

As Tabelas 4.3 e 4.4 mostram os campos das tabelas *te_graph_algorithms* e *te_graph_algorithms_params*.

- algol_id: identificador único do algoritmo;
- name: nome do algoritmo;
- description: descrição sobre o funcionamento do algoritmo.

Tabela 4.3 - Tabela *te_graph_algorithms*.

Nome	Tipo	Não é NULL	Chave
algol_id	INTEGER	Sim	Sim
name	VARCHAR	Não	Não
description	VARCHAR	Não	Não

Tabela 4.4 - Tabela *te_graph_algorithms_params*.

Nome	Tipo	Não é NULL	Chave
algol_id	INTEGER	Sim	Sim
parameter_number	INTEGER	SIM	Sim
parameter_name	VARCHAR	Não	Não
parameter_desc	VARCHAR	Não	Não

- algol_id: identificador único do algoritmo;
- parameter_number: identificação do parâmetro em relação ao algoritmo;
- parameter_name: nome do parâmetro;
- parameter_desc: descrição do parâmetro.

Os comandos *SQL* necessários para a construção destas tabelas são apresentados abaixo.

```
CREATE TABLE te_graph_algorithms (
  algol_id serial NOT NULL,
  name VARCHAR NULL,
  description VARCHAR NULL,
  PRIMARY KEY(algol_id)
);

CREATE TABLE te_graph_algorithms_params (
  algol_id serial NOT NULL,
  parameter_number serial NOT NULL,
  parameter_name VARCHAR NULL,
  parameter_description VARCHAR NULL,
  PRIMARY KEY(algol_id, parameter_number)
);
```

4.2 Modelo de Dados

O modelo de dados utilizado para a representação de um grafo utiliza os conceitos de orientação a objeto. É definido um conjunto de classes que representam cada elemento de um grafo, tornando mais fácil sua utilização, além de auxiliar na utilização de bibliotecas de terceiros.

É possível definir um modelo flexível através da definição abstrata das principais operações; funções de inserção, remoção e acesso aos elementos são definidos em uma classe virtual representando o modelo genérico de um grafo.

A seguir é mostrado o diagrama de classes dos principais componentes (Fig. 4.2).

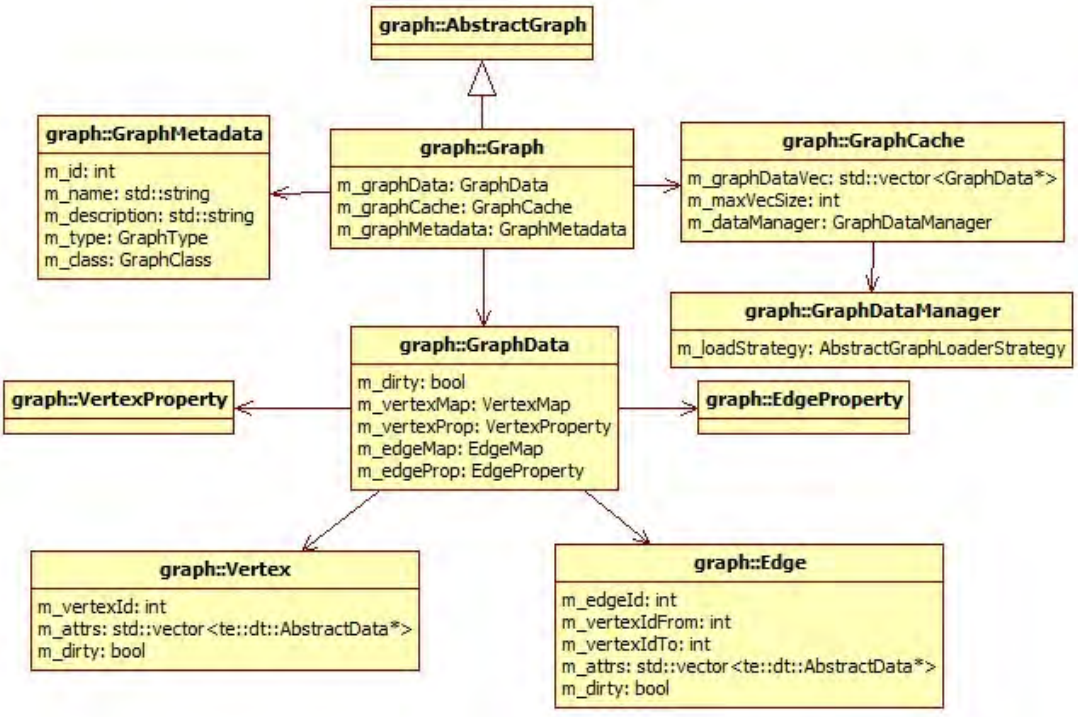


Figura 4.2 - Modelo de classes proposto para representação de um grafo.

- *AbstractGraph*: definição da interface abstrata de um grafo genérico;
- *Graph*: implementação concreta de um grafo que acessa os dados em um SGBD e utiliza a memória para *cache*;
- *GraphMetadata*: conjunto de atributos que descrevem o grafo;

- *Vertex*: objeto que representa um vértice;
- *VertexProperty*: utilizado para indicar quais atributos estão sendo associados aos vértices;
- *Edge*: objeto que representa uma aresta;
- *EdgeProperty*: utilizado para indicar quais atributos estão sendo associados às arestas;
- *GraphData*: representação de um conjunto de dados (vértices e arestas);
- *GraphCache*: estrutura utilizada para realização de *cache*;
- *GraphDataManager*: gerenciador de dados utilizado pelo *cache*.

Uma outra forma de vermos o relacionamento entre essas classes é atraves da Figura 4.3.

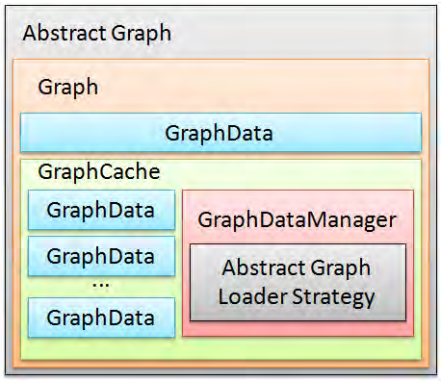


Figura 4.3 - Relacionamento entre as classes.

A seguir é apresentado uma descrição mais detalhada de cada classe definida no modelo.

4.2.1 *AbstractGraph*

Definindo-se uma interface abstrata que represente o grafo apenas com suas funções mais comuns e posteriormente criando especializações, permite definir estruturas mais otimizadas. É bem provavel que não seja possível definir uma estrutura única de dado que represente todas as funcionalidades de um grafo. Ao criarmos essas especializações é possível definirmos objetos mais inteligentes e otimizados.

Algumas das principais funções virtuais da classe *AbstractGraph* são:

- Funções de controle dos vértices

  virtual void add(Vertex* v) = 0;

  virtual void update(Vertex* v) = 0;

  virtual te::graph::Vertex* getVertex(const int id) = 0;

  virtual void removeVertex(const int id) = 0;

- Funções de iteração e travessia dos vértices

  virtual te::graph::Vertex* getFirstVertex() = 0;

  virtual te::graph::Vertex* getNextVertex() = 0;

  virtual te::graph::Vertex* getPreviousVertex() = 0;

- Funções de atributos dos vértices

  virtual void addVertexProperty(te::dt::Property* p) = 0;

  virtual void removeVertexProperty(int idx) = 0;

  virtual te::dt::Property* getVertexProperty(int idx) = 0;

- Funções de controle das arestas

  virtual void add(Edge* e) = 0;

  virtual void update(Edge* e) = 0;

  virtual te::graph::Edge* getEdge(const int id) = 0;

  virtual void removeEdge(const int id) = 0;

- Funções de iteração e travessia das arestas

  virtual te::graph::Edge* getFirstEdge() = 0;

  virtual te::graph::Edge* getNextEdge() = 0;

  virtual te::graph::Edge* getPreviousEdge() = 0;

- Funções de atributos das arestas

  virtual void addEdgeProperty(te::dt::Property* p) = 0;

  virtual void removeEdgeProperty(int idx) = 0;

  virtual te::dt::Property* getEdgeProperty(int idx) = 0;

Para o caso de grafos direcionados temos funções do tipo:

- Funções que necessitam de informação de orientação.

  virtual bool isSourceVertex(const int id) = 0;

  virtual bool isSinkVertex(const int id) = 0;

  virtual std::vector<te::graph::Edge*> getInEdges(const int vId) = 0;

  virtual std::vector<te::graph::Edge*> getOutEdges(const int vId) = 0;

Algumas da extensões da classe *AbstractGraph* e seus respectivos métodos para acesso as arestas são (Fig.4.4):

- *Grafo Não direcional*: não existe a distinção das arestas que entram ou saem de um vértice, as arestas são tratadas como incidentes.
  - std::vector<te::graph::Edge*> getEdges(const int vId)
- *Grafo Direcional*: as arestas possuem direção, neste caso os vértices possuem as informações distintas das arestas que saem de seus vértices.
  - std::vector<te::graph::Edge*> getOutEdges(const int vId)
- *Grafo Bidirecional*: as arestas possuem direção, neste caso os vértices possuem as informações distintas das arestas que saem e chegam em seus vértices.
  - std::vector<te::graph::Edge*> getOutEdges(const int vId)
  - std::vector<te::graph::Edge*> getInEdges(const int vId)

Os diferentes tipos de extensão da classe abstrata existem para que a estrutura de dados a ser criada para representar o grafo seja a mais adequada em cada caso, isso faz com que apenas dados úteis estejam presentes na memória, também evitando que processamentos desnecessários sejam feitos. Por exemplo, em uma aplicação em que a direção da aresta não é fundamental, não há motivo em se utilizar uma estrutura bidirecional em que existe a necessidade de se criar dois vetores auxiliares para cada vértice para armazenar essa informação de direção, sem mencionar o custo computacional para se manter atualizado essas informações.

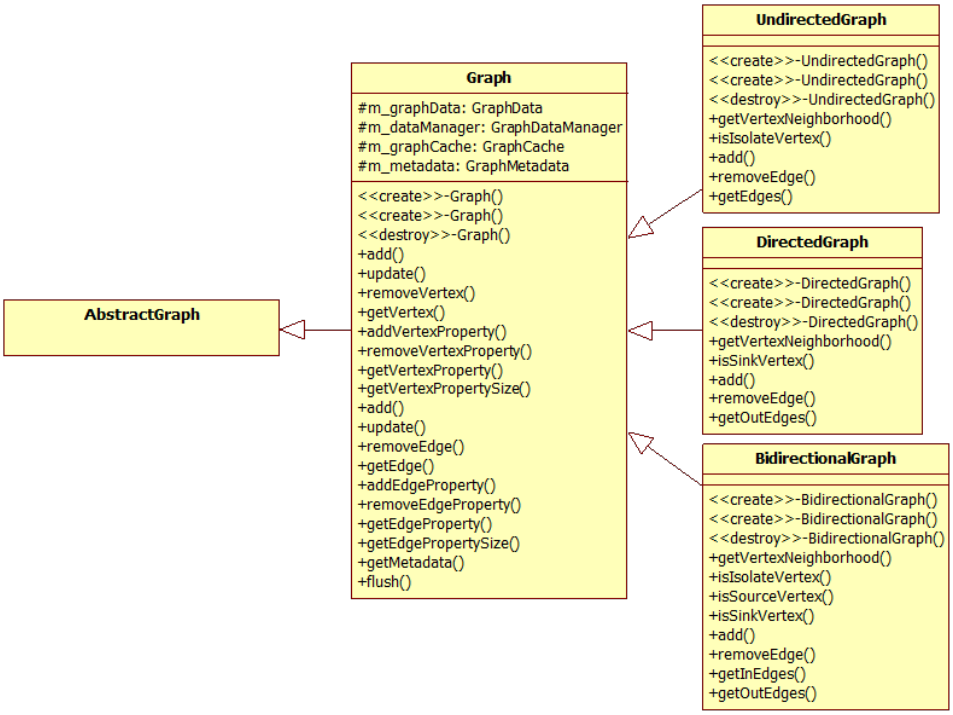


Figura 4.4 - Diagrama de classes dos tipos de grafos implementados.

4.2.2 *Vertex*

A classe *Vertex* representa os vértices de um grafo e é composto pelos seguintes atributos:

- *m_vertexId*: atributo utilizado para identificar o vértice;
- *m_edges*: possui a informação de vizinhança, identificação de cada aresta incidente (utilizado em casos de grafos não direcionados);
- *m_edgesIn*: possui a informação de vizinhança, identificação de cada aresta que chega a este vértice (utilizado em casos de grafos direcionados);
- *m_edgesOut*: possui a informação de vizinhança, identificação de cada aresta que sai deste vértice (utilizado em casos de grafos direcionados);
- *m_dirty*: atributo utilizado para indicar que o vértice teve alterações e deve ser atualizado na base de dados;
- *m_attrs*: este atributo é um container de objetos utilizados para associar dados aos vértices; esses dados podem ser valores numéricos, literais ou

espaciais.

A classe *VertexProperty* é utilizada para armazenar as informações que descrevem os atributos associados a cada vértice. Funciona como um dicionário de dados para o atributo *m_attrs* da classe *Vertex*. Esta é uma abordagem interessante pois evita o excesso de repetição dessas informações em cada vértice.

4.2.3 *Edge*

A classe *Edge* representa o relacionamento entre dois vértices e é composto pelos seguintes atributos:

- *m_edgeId*: atributo utilizado para identificar as arestas;
- *m_vertexIdFrom*: atributo utilizado para identificar o vértice de origem;
- *m_vertexIdTo*: atributo utilizado para identificar o vértice de destino;
- *m_dirty*: atributo utilizado para indicar que a aresta teve alterações e deve ser atualizada na base de dados;
- *m_attrs*: este atributo é um container de objetos utilizados para associar dados às arestas; esses dados podem ser valores numéricos, literais ou espaciais.

A classe *Edge* também possui uma classe auxiliar para representar as informações a respeito de seus atributos armazenados em *m_attrs*. Essa classe é chamada *EdgeProperty*.

4.3 Acesso aos dados em memória

O acesso aos elementos do grafo definidos na interface abstrata não implica onde esses dados devem estar armazenados. Uma abordagem feita neste trabalho foi de armazená-los em bancos de dados relacionais, mas sempre tendo uma parte desses dados em memória para otimizar seu acesso.

Estruturas auxiliares, mostradas a seguir, foram criadas para ajudar neste processo.

4.3.1 *GraphData*

A classe *GraphData* foi projetada com a finalidade de agrupar conjuntos de dados (vértices e arestas) criando o conceito de pacotes. Essa metodologia tem como

finalidade auxiliar na gerência dos dados em memória, facilitando sua busca e armazenamento.

Essa classe é composta de:

- *m_vertexMap*: conjunto de vértices;
- *m_edgeMap*: conjunto de arestas;
- *m_dirty*: atributo utilizado para indicar que este pacote teve alterações e deve ser atualizado na base de dados.

O tamanho desses cojuntos é configurável. No arquivo de configuração *Config.h* é possível definir esse valor.

```
#define TE_GRAPH_MAX_CACHE_SIZE 10000
```

4.4 Acesso aos dados no repositório

Quando um grafo esta sendo acessado pela primeira vez ou quando um elemento do grafo desejado ainda não se encontra em memória ou em *cache*, é necessário fazer a sua carga (Fig. 4.5).

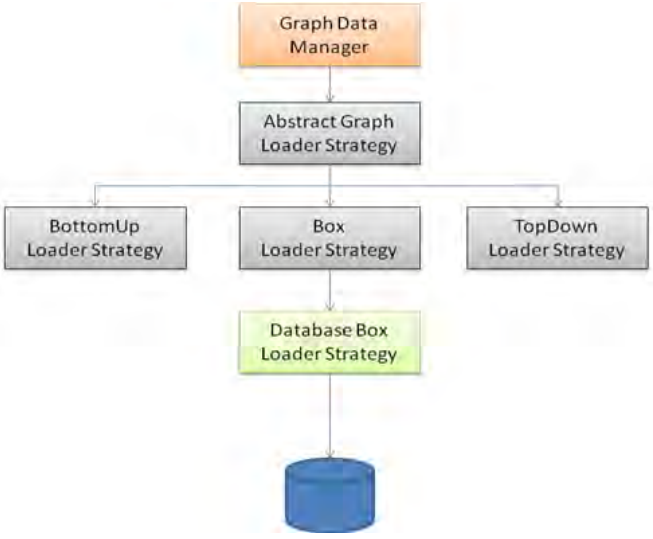


Figura 4.5 - Esquema para recuperação de dados do repositório.

Esse processo de carregar os dados para a memória não ocorre de elemento a elemento. Um conjunto de dados selecionados seguindo um padrão é recuperado e ar-

mazenados em pacotes (*GraphData*) que ficam em memória. As funções para realizar esse tipo de operação são definidas na classe *GraphDataManager*.

4.4.1 *GraphDataManager*

A classe *GraphDataManager* serve como uma fachada para o acesso dos dados em seu repositório. Esta classe possui um único atributo, classe *AbstractGraphLoaderStrategy* e um conjunto de funções virtuais:

- GraphData* loadGraphDataByVertexId(const int& vertexId);
- GraphData* loadGraphDataByEdgeId(const int& edgeId);
- void saveGraphData(GraphData* data);

Esta classe não implementa nenhuma dessas operações, apenas irá repassar a chamada para uma função equivalente de uma implementação concreta da classe *AbstractGraphLoaderStrategy*.

4.4.2 *AbstractGraphLoaderStrategy*

A classe *AbstractGraphLoaderStrategy* define o conceito de *LoaderStrategy* que determina como um dado deve ser carregado. Algumas estratégias propostas foram definidas (Fig. 4.6):

- *BottomUp*: carrega um conjunto de objetos a partir do elemento procurado, percorrendo o grafo de forma reversa;
- *TopDown*: carrega um conjunto de objetos a partir do elemento procurado, percorrendo o grafo seguindo seu fluxo normal;
- *Box*: carrega um conjunto de objetos tendo como objeto central o elemento procurado (implementado);
- *Sequence*: carrega um conjunto de objetos de forma sequencial, tendo como objeto inicial o elemento procurado (implementado).

Para acessar os dados de um grafo estando armazenados em um banco de dados relacional, utilizando o modelo de arestas para seu armazenamento, a recuperação é feita do seguinte modo:

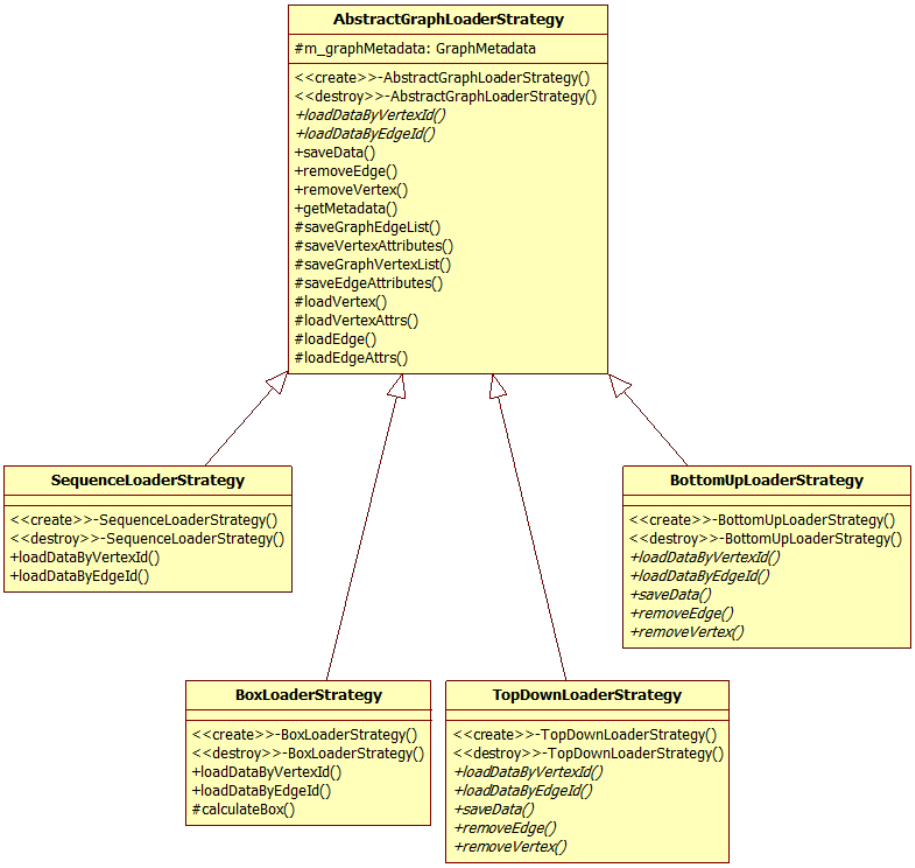


Figura 4.6 - Estratégias para busca de dados na fonte de dados.

Definie-se uma SQL que recupera em um único passo as informações das arestas e vértices.

```
SELECT * FROM teste_model_edge AS edges
JOIN teste_attr_model_vertex AS v1
        ON (edges.vertex_from = v1.vertex_id)
JOIN teste_attr_model_vertex AS v2
        ON (edges.vertex_to = v2.vertex_id)
```

A cláusula *Where* desta SQL é definida dependendo de qual estratégia se está utilizando, seja por *Box*, *Query* ou *Sequence*. A forma de como o banco responde a esta pesquisa é mostrada na Figura 4.7.

O resultado de uma pesquisa em um banco de dados relacional é uma outra tabela. Neste caso estamos fazendo uma pesquisa em duas tabelas distintas:

- *teste_model_edge*: contendo informações sobre as arestas e seus atributos;

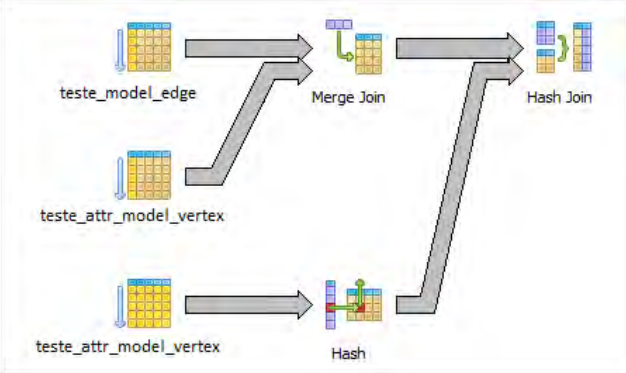


Figura 4.7 - Fluxograma de uma pesquisa no banco de dados.

- *teste_attr_model_vertex*: contendo informações sobre os vértices e seus atributos.

A tabela resultante irá possuir as informações das arestas e informações tanto dos vértices de origem quanto de destino. Isso é possível através de uma operação de junção entre as tabelas devido ao fato de os vértices serem representados por um identificador único.

4.5 Cache

Uma parte importante deste trabalho e que irá auxiliar no desempenho de acesso aos elementos do grafo é a estrutura de *cache*. Podemos entender o *cache* como sendo o histórico de tudo que já foi acessado. Se tivermos um bom tamanho desse histórico e quanto mais inteligente ele for ao descartar informações, melhor proveito ele terá. Fazendo uma simples analogia, podemos entender como a classe *Graph* sendo o processador principal de um computador onde se tem acesso direto e muito rápido a um pequeno conjunto de elementos, a classe *GraphCache* sendo a memória RAM tendo uma parte dos dados carregada e com um acesso não tão rapido e por último a classe *GraphDataManager* sendo nossa fonte de dados com um acesso lento (Fig. 4.8).

4.5.1 *GraphCache*

A classe *GraphCache* define como os dados são temporariamente armazenados em memória, sendo composta pelos seguintes atributos:

- m_graphDataMap: container com todos os *GraphData* já carregados;

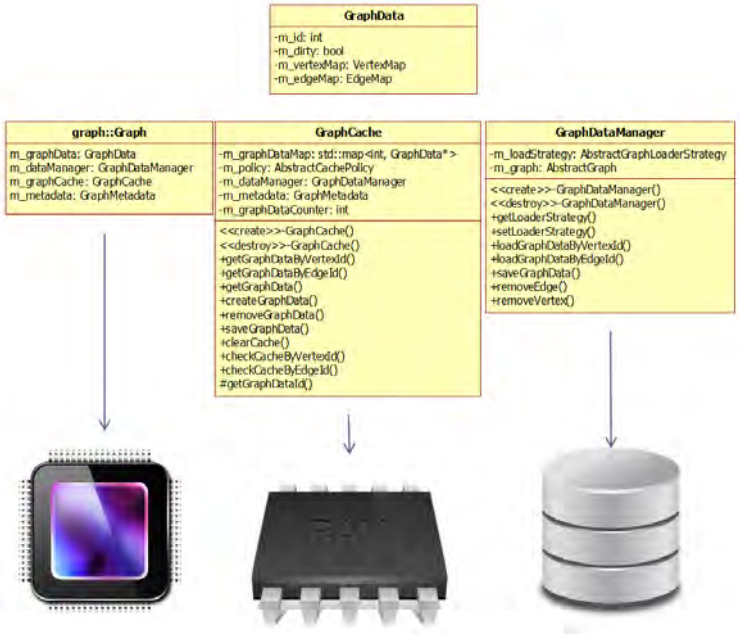


Figura 4.8 - Esquema de utilização do *cache*.

- m_dataManager: instância da classe *GraphDataManager* que irá acessar os dados no repositório;

- m_policy: instância da classe *AbstractCachePolicy* que define a forma de policiamento dos dados na memória.

O tamanho do container para armazenar os dados em memória é configurável. No arquivo de configuração *Config.h* é possível definir esse valor.

```
#define TE_GRAPH_MAX_VEC_CACHE_SIZE 100
```

Duas observações importantes a fazer a respeito dessa estrutura de *cache* são:

- Tamanho do vetor: quanto mais pacotes o cache possuir, maiores são as chances de o elemento procurado estar carregado, porém em cada busca ele terá que pesquisar em mais pacotes;

- Política de *cache*: uma vez atingido o limite máximo de pacotes é necessário a remoção de pacotes da memória. Duas estratégias de policiamento são definidas:

– *FIFO (First In First Out)*: conceito de uma fila comum, o primeiro pacote a entrar será o primeiro pacote a sair (DEITEL; DEITEL, 2002);

– *LFU (Least Frequency Used)*: pode ser visto como uma fila também, só que a cada vez que um pacote é usado ele é movido para o começo da fila e as remoções são feitas no fim da fila (DEITEL; DEITEL, 2002).

Essa política passa a ser simples devido ao fato de os elementos estarem agrupados em pacotes. O controle é feito por pacotes e não por elementos individuais, evitando uma sobrecarga de checagens e controles (Fig. 4.9).

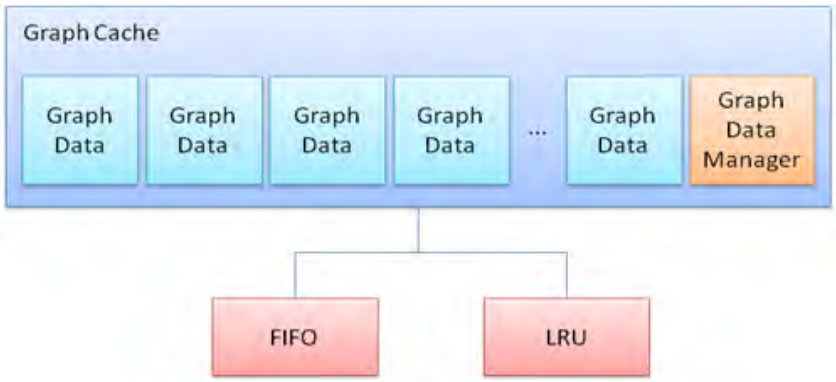


Figura 4.9 - Esquema de utilização do *cache*.

4.6 *Iterators*

Iterador é um padrão de projeto comportamental que pode ser definido como objetos que permitem a travessia genérica de estruturas de dados sem expor sua real representação (GAMMA et al., 1994). Esse tipo de operação sobre grafos é bastante válida pois permite o acesso a todos os elementos independente de sua topologia. Outra característica interessante dos iteradores é a possibilidade de criarmos estratégias diferentes para seu percorrimento, possibilitando acessar conjuntos restritos dos dados. Por exemplo, no caso da visualização de um grafo, é necessário ser possível acessar apenas os elementos de uma determinada área. Outro exemplo seria acessarmos apenas os elementos que tenham uma característica específica.

Para que esse percorrimento genérico da estrutura de dados do grafo fosse possível foi criada uma interface abstrata de um iterador que define virtualmente suas principais funções e posteriormente foram criadas especializações permitindo que diversas opções de percorrimento dos dados fosse possível. Como o grafo é formado

por duas estruturas de dados distintas (vértices e arestas), é possível a travessia dos dados seja pelos vértices, seja pelas arestas. A classe abstrata do iterador é chamada *AbstractIterator* e tem as seguintes funções virtuais:

- virtual te::graph::Vertex* getFirstVertex() = 0;
- virtual te::graph::Vertex* getNextVertex();
- virtual te::graph::Vertex* getPreviousVertex();
- virtual te::graph::Edge* getFirstEdge() = 0;
- virtual te::graph::Edge* getNextEdge();
- virtual te::graph::Edge* getPreviousEdge();

As especializações implementadas foram:

- *Sequence*: realiza a travessia de todos os elementos do grafo;
- *Box*: realiza a travessia dos elementos de uma área em específico;
- *Query*: realiza a travessia dos elementos dada uma restrição.

Na Figura 4.10 é mostrado o diagrama de classe dos iteradores implementados.

4.7 *SQL's Genéricas*

No desenvolvimento deste trabalho foi necessário a definição de um conjunto de SQL's para auxiliar na persistência e carga dos metadados e dados dos grafos. Com a adoção da *TerraLib* como ambiente de desenvolvimento e seguindo seu padrão de codificação, isso foi alterado. A *TerraLib 5* possui um módulo chamado *DataAccess* no qual define os conceitos de *Data Source*, *Data Set* e *Query* genéricos, permitindo que um dado seja acessado nas mais diversas fontes de dados. Para que isso seja possível foram criadas abstrações para cada função definida pela linguagem SQL.

Essa característica se mostrou um grande ganho para este trabalho pois permite que os dados e metadados dos grafos possam estar em outras fontes de dados que não sejam os bancos de dados relacionais.

Uma SQL para busca dos dados que antes era definida da seguinte forma:

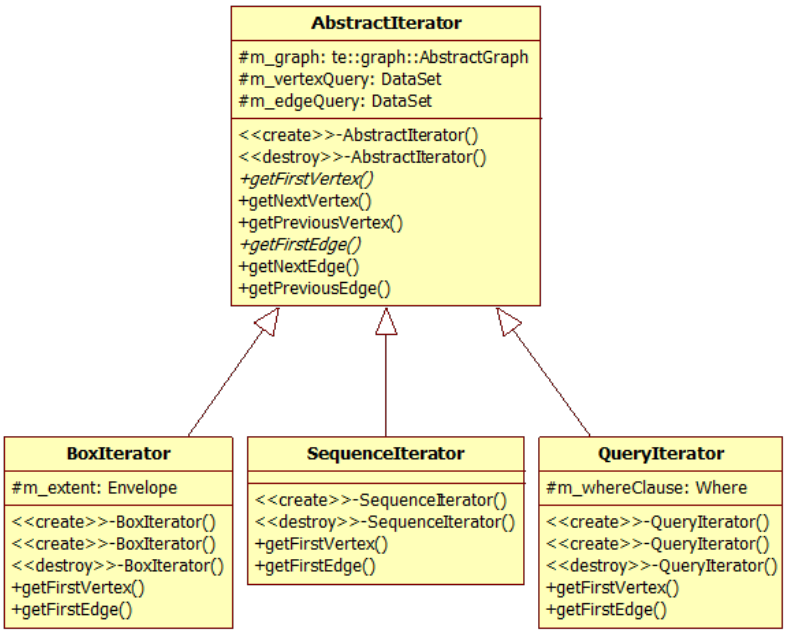


Figura 4.10 - Diagrama de classe dos iteradores.

```
SELECT * FROM teste_model_edge AS edges
JOIN teste_attr_model_vertex AS v1
        ON (edges.vertex_from = v1.vertex_id)
JOIN teste_attr_model_vertex AS v2
        ON (edges.vertex_to = v2.vertex_id)
```

Passa a ser definida assim:

```
//fields
te::da::Fields* all = new te::da::Fields;
all->push_back(new te::da::Field("*"));

//from
te::da::From* from = new te::da::From;

te::da::FromItem* fi1 =
  new te::da::DataSetName(edgeTable, "edges");
from->push_back(fi1);
te::da::FromItem* fi2 =
  new te::da::DataSetName(vertexAttrTalbe, "v1");
from->push_back(fi2);
te::da::FromItem* fi3 =
  new te::da::DataSetName(vertexAttrTalbe, "v2");
```

```
from->push_back(fi3);

//where
std::string vertexFrom = "edges.";
  vertexFrom += Globals::sm_tableEdgeModelAttrVFrom;
std::string vertexTo = "edges.";
  vertexTo += Globals::sm_tableEdgeModelAttrVTo;
std::string v1Id = "v1.";
  v1Id += Globals::sm_tableVertexModelAttrId;
std::string v2Id = "v2.";
  v2Id += Globals::sm_tableVertexModelAttrId;

te::da::Field* fvf = new te::da::Field(vertexFrom);
te::da::Field* fv1id = new te::da::Field(v1Id);
te::da::Expression* exp1 =
new te::da::EqualTo(fvf->getExpression(),
  fv1id->getExpression());

te::da::Field* fvt = new te::da::Field(vertexTo);
te::da::Field* fv2id = new te::da::Field(v2Id);
te::da::Expression* exp2 =
  new te::da::EqualTo(fvt->getExpression(),
  fv2id->getExpression());

te::da::And* and = new te::da::And(exp1, exp2);

te::da::Where* wh = new te::da::Where(and);

//select
te::da::Select select(all, from, wh);
```

Mesmo que isso gere uma complexidade maior na programação, o resultado é uma função totalmente independente da fonte de dado.

4.8 Extratores

Os extratores são funções capazes de gerar um grafo a partir de um processamento sobre uma base de dados, como por exemplo, dados vetoriais, imagens e mesmo tabelas de dados. A seguir são apresentados os extratores implementados.

4.8.1 LDD

Um LDD (*Local Drain Directions*) é uma matriz com valores bem defindos, resultado de um processo de extração de fluxos locais em um MNT (Modelo Numérico de Terreno) que indicam a direção do fluxo em cada *pixel* (ROSIM, 2008).

Percorrendo a matriz do LDD é possível extrair o grafo representando os fluxos. Cada *pixel* será um vértice do grafo e as arestas serão geradas baseadas no valor do *pixel*. A Tabela 4.5 indica os valores presentes no LDD e qual a direção que cada um representa.

Tabela 4.5 - Valores e direções de um LDD.

32 ↖	64 ↑	128 ↗
16 ←		1 →
8 ↙	4 ↓	2 ↘

As imagens representando um modelo numérico de terreno e o LDD que o representa é apresentado na Figura 4.11. O resultado da extração do grafo é apresentado na seção 4.8.1.

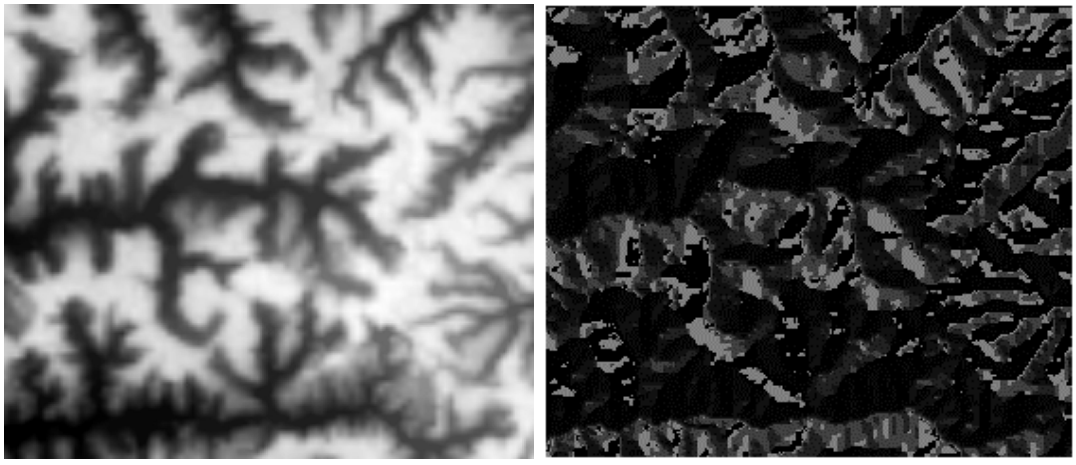


Figura 4.11 - Imagens: (a)MNT, (b) LDD

4.8.2 RAG

RAG (*Region Adjacency Graphs*) é uma estrutura de grafo que representa a visão espacial de uma imagem, associando um vértice a cada região e uma aresta a cada

par de regiões adjacentes, Figura 4.12. A associação de valores às arestas pode variar de acordo com cada aplicação, mas em geral esses valores são definidos como medidas de similaridades. Korting (2007) adotou os seguintes valores de similaridade para as arestas:

- -1 para segmentos não adjacentes;
- 0 para segmentos adjacentes de classes diferentes;
- 1 para segmentos adjacentes de mesma classe.

O grafo é extraído a partir de uma imagem rotulada (resultado de um processo de segmentação), ou mesmo a partir de um conjunto de polígonos. A Figura 4.12 representa a extração de um grafo.

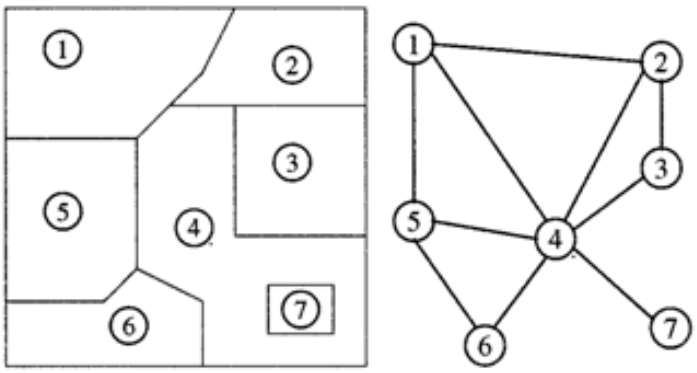


Figura 4.12 - Representação de um grafo a partir de uma região de adjacência.

Exemplos de grafos extraídos utilizando esse método é apresentado na seção 4.8.2.

4.8.3 Flow

Fluxo pode ser definido e entendido de diversas formas, mas em geral representa o deslocamento de pessoas ou objetos de um local A para um local B. Oliveira (2005) em seu trabalho utiliza das redes de atenção hospitalar para representar os fluxos de pacientes e assim estudar o acesso aos serviços hospitalares do Sistema Único de Saúde no Brasil.

A extração do grafo é feita através de um dado vetorial contendo a representação geométrica da área em estudo, e uma tabela de dados representando os fluxos, sendo

que essa tabela deve ser composta por colunas que indiquem a origem e o destino do fluxo. A Figura 4.13 mostra como é um grafo representando os fluxos.

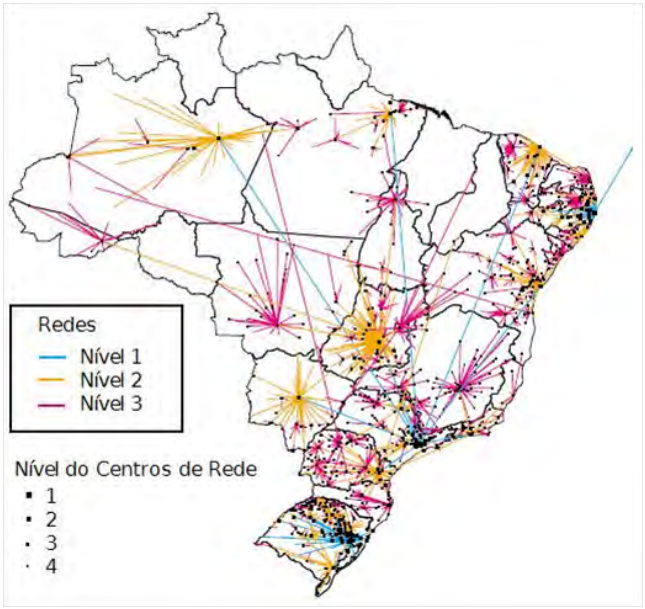


Figura 4.13 - Representação de um grafo extraído a partir de informações de fluxo. Fonte: Oliveira (2005).

Um exemplo de extração utilizando esse método é apresentado na seção 4.8.3.

4.8.4 Graph Framework

Para que este trabalho fosse realizado foi necessário a utilização da biblioteca *TerraLib* para suporte no tratamento aos dados espaciais, além das funções para acesso às diversas fontes de dados. Essa possibilidade de acessar diversas fontes de dados de forma transparente permite a este trabalho ser independente de qualquer fonte de dados.

Toda a codificação feita neste trabalho segue o modelo de programação adotado pela *TerraLib*. Uma outra característica deste projeto foram os diversos padrões de projeto (GAMMA et al., 1994) utilizados em seu desenvolvimento (fábricas, estratégias, iteradores).

O conjunto de classes representando os elementos de um grafo, bem como as estratégias de *cache*, busca dos dados e iteradores, formam um *framework* para manipulação de grafos dentro do ambiente *TerraLib*.

Este *framework* foi todo escrito utilizando-se a linguagem de programação C++. A Figura 4.14 apresenta o número de arquivos e quantas linhas de códigos foram criadas.

```
http://cloc.sourceforge.net v 1.56  T=1.0 s (88.0 files/s, 12790.0 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C++                             42           1550            778           4372
C/C++ Header                    46           1916           2709           1465
-------------------------------------------------------------------------------
SUM:                            88           3466           3487           5837
-------------------------------------------------------------------------------
```

Figura 4.14 - Relatório dos arquivos e linhas de código do *framework* de grafos.

Os arquivos criados para este *framework* se encontram organizados da seguinte forma (Fig. 4.15).

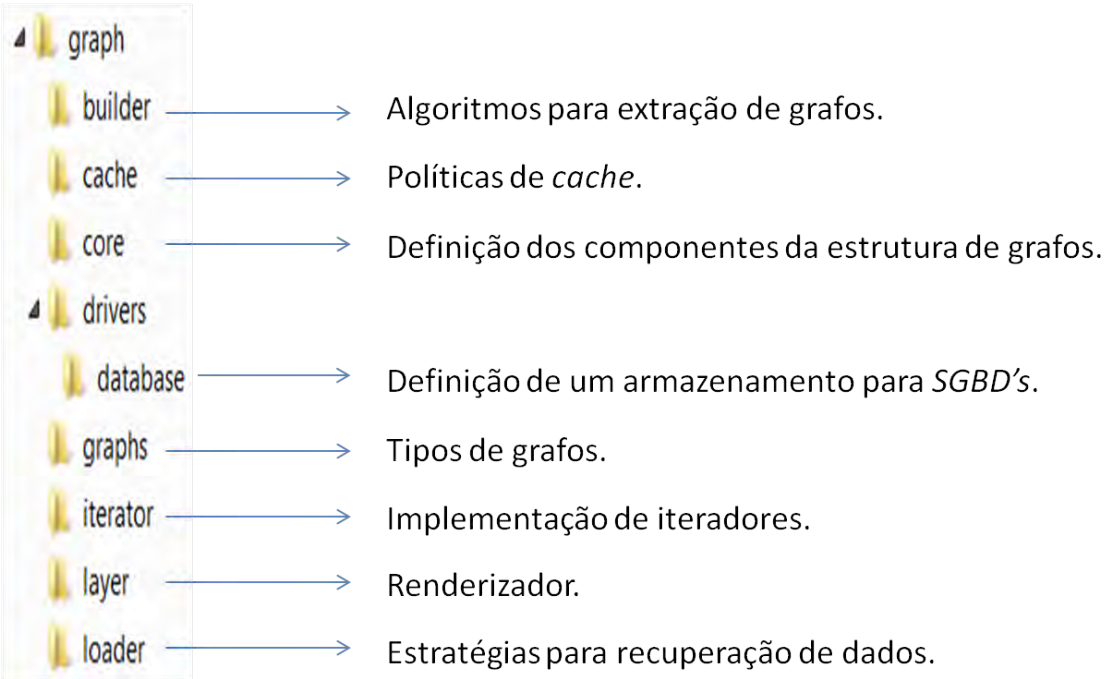


Figura 4.15 - Organização de diretórios dos arquivos deste *framework*.

5 RESULTADOS

Neste capítulo são demonstrados alguns resultados obtidos, bem como a utilização da API desenvolvida. Em alguns dos exemplos citados que apresentam trechos de códigos mostrando as funcionalidades deste *framework* são utilizados tipos de dados nativos da *TerraLib*, sendo que informações a respeito podem ser encontradas no endereço (http://www.dpi.inpe.br/terralib5/wiki).

5.1 Multi Pacotes

Umas das opções disponíveis para a utilização da API desenvolvida é a estratégia de *cache* utilizando pacotes. Através da configuração de variáveis é possível definir o número de pacotes em memória, bem como o número de elementos presentes em cada pacote.

Essas variáveis são:

- TE_GRAPH_MAX_CACHE_SIZE: Número de elementos em cada pacote.
- TE_GRAPH_MAX_VEC_CACHE_SIZE: Número possível de pacotes em memória.

Essa metodologia de agrupar os dados em pacotes *GraphData* foi definida para simplificar as estratégias de *cache*. Partindo do princípio que não é possível alocar um grafo inteiro na memória, é necessário definir estratégias para manter em memória apenas uma parte dos dados. Esse controle pode ser feito elemento a elemento, o que iria gerar um excesso de verificações e a adição de mais informações em cada elemento, ou pode-se agrupar os elementos em pacotes, simplificando essas verificações; a segunda opção foi a adotada.

Essa solução de multi pacotes associado com as estratégias de *cache* solucionaram o principal problema deste trabalho que era o tratamento de grandes grafos.

Adotando esta solução, a dúvida agora seria como definir o número ideal de pacotes em memória e a quantidade de elementos em cada um. Foi criado um exemplo para que se pudesse analizar o desempenho dessa estratégia variando esses dois fatores.

No experimento, foi realizado o acesso aleatório a elementos de um grafo em memória contendo os seguintes números de elementos:

- vértices: 68.913 elementos;

- arestas: 64.258 elementos.

Para o acesso aleatório de 1.000.000 (um milhão) de elementos, os resultados podem ser vistos na Tabela 5.1.

Tabela 5.1 - Resultado do uso da estratégia de multi pacotes.

Cache Size	*Vector Cache Size*	Tempo
1.000	100 (68)	63 segundos
10.000	100 (6)	11 segundos
100.000	100 (1)	2 segundos

O trecho de código necessário para realizar este exemplo pode ser visto abaixo.

```
#define NUMBER_OF_ITERACTIONS 1000000

void GetGraphElements(te::graph::AbstractGraph* graph,
        te::rst::Raster* raster)
{
  int nCols = raster->getNumberOfColumns();
  int nRows = raster->getNumberOfRows();

  srand(time(NULL));

  for(unsigned int i = 0; i < NUMBER_OF_ITERACTIONS; ++i)
  {
    int curCol = rand() % nCols;
    int curRow = rand() % nRows;

    int index = (curRow * nCols) + curCol;

    te::graph::Vertex* v = graph->getVertex(index);

    if(!v)
    {
      std::cout << "Error getting vertex." << std::endl;
    }
  }
}
```

Percebe-se que ao se utilizar pacotes com mais elementos, a velocidade de acesso aos elementos tem uma melhora significativa. Isso não quer dizer que se for utilizado

apenas um pacote seria obtido um desempenho melhor, pois neste caso o *cache* não faria sentido.

Este é um teste de acesso a elementos de forma aleatória; em casos reais o acesso seria sequencial (seguindo a travessia do grafo), e em grande parte das vezes o elemento estaria dentro de um mesmo pacote; assim, a busca no *cache* por outros pacotes seria bem menos frequente.

O que se tem que levar em consideração deste exemplo é que para se decidir o número de elementos de um pacote tem-se que levar em consideração outros fatores, bem como qual é a melhor estratégia de busca de dados em cada caso.

5.2 Interface e manipulação de grafos

Como um dos objetivos deste trabalho é a criação de um *framework* para a manipulação de grafos, é importante mostrar alguns exemplos de como esses objetos que representam as estruturas de grafos, *cache* e iteradores podem ser utilizados.

5.2.1 Create

O primeiro exemplo é de como pode-se criar um objeto do tipo grafo. Para que isso seja possível é necessário decidir e informar alguns parâmetros, tais como:

- Onde o grafo será armazenado;
- Como o dado será armazenado;
- Qual a forma de carregar os dados armazenados;
- Que tipo de estratégia de *cache* será utilizada.

Todas essas informações são definidas como um conjunto de parâmetros. A partir dessas informações o *framework* saberá criar corretamente o tipo de grafo. No exemplo a seguir, são definidos os seguintes parâmetros de criação:

- Banco POSTGIS como repositório de dados;
- O grafo será armazenado como lista de arestas;
- Estratégia sequencial para busca dos dados no repositório;
- Definida a política **FIFO** para controle do *cache*.

```
// data source information
std::map<std::string, std::string> connInfo;
connInfo["host"] = "localhost";
connInfo["user"] = "postgres";
connInfo["password"] = "abcde";
connInfo["dbname"] = "t5graph";
connInfo["connect_timeout"] = "4";

// graph type
std::string graphType =
  te::graph::Globals::sm_graphFactoryDefaultObject;

// graph information
std::map<std::string, std::string> graphInfo;
graphInfo["GRAPH_DATA_SOURCE_TYPE"] = "POSTGIS";
graphInfo["GRAPH_NAME"] = "teste";
graphInfo["GRAPH_DESCRIPTION"] = "Exemplo de utilizacao";
graphInfo["GRAPH_STORAGE_MODE"] =
  te::graph::Globals::sm_edgeStorageMode;
graphInfo["GRAPH_STRATEGY_LOADER"] =
  te::graph::Globals::sm_factoryLoaderStrategyTypeSequence;
graphInfo["GRAPH_CACHE_POLICY"] = "FIFO";

//create output graph
te::graph::AbstractGraph * graph =
  te::graph::AbstractGraphFactory::make(graphType, connInfo, graphInfo);
```

5.2.2 Open

Para se acessar um grafo já armazenado em alguma fonte de dados, o processo é semelhante; também é necessário a definição de um conjunto de parâmetros que indiquem como e onde ele está armazenado e como o grafo deve ser manipulado em memória. Além desses parâmetros, é necessário definir a identificação do grafo a ser carregado, como no exemplo abaixo.

```
// data source information
std::map<std::string, std::string> connInfo;
connInfo["host"] = "localhost";
connInfo["user"] = "postgres";
connInfo["password"] = "abcde";
connInfo["dbname"] = "t5graph";
connInfo["connect_timeout"] = "4";

// graph type
```

```
std::string graphType =
  te::graph::Globals::sm_graphFactoryDefaultObject;

// graph information
std::map<std::string, std::string> graphInfo;
graphInfo["GRAPH_DATA_SOURCE_TYPE"] = "POSTGIS";
graphInfo["GRAPH_ID"] = "1";  /* Graph Identifier */
graphInfo["GRAPH_STORAGE_MODE"] =
  te::graph::Globals::sm_edgeStorageMode;
graphInfo["GRAPH_STRATEGY_LOADER"] =
  te::graph::Globals::sm_factoryLoaderStrategyTypeSequence;
graphInfo["GRAPH_CACHE_POLICY"] = "FIFO";

//open graph
te::graph::AbstractGraph* graph =
  te::graph::AbstractGraphFactory::open(graphType,
        connInfo, graphInfo);
```

5.2.3 Add Elements

Um exemplo trivial de utilização é a adição de elementos (vértices e arestas) em um grafo. Este exemplo também mostra a criação de uma nova propriedade para ser associada aos vértices, permitindo que um novo tipo de informação (neste caso um atributo geométrico) possa ser informado para cada elemento.

```
//create graph attribute
te::gm::GeometryProperty* gProp =
  new te::gm::GeometryProperty("coords");
gProp->setGeometryType(te::gm::PointType); /* Point type */
gProp->setSRID(...);
graph->addVertexProperty(gProp);

// create vertex 0
Vertex* v0 = new Vertex(0);
v0->setAttributeVecSize(graph->getVertexPropertySize());
te::gm::Point* p0 = new te::gm::Point(...);
v0->addAttribute(0, p0);
graph->add(v0);

//create vertex 1
Vertex* v1 = new Vertex(1);
v1->setAttributeVecSize(graph->getVertexPropertySize());
te::gm::Point* p1 = new te::gm::Point(...);
v1->addAttribute(0, p1);
```

```
graph->add(v1);

//create edge
Edge* e = new Edge(0, 0, 1);
graph->add(e);
```

Neste exemplo foram criados dois vértices com identificações *0* e *1*, e uma aresta composta por esses dois vértices, com o identificador *0*.

5.2.4 Iterators

A utilização de iteradores é muito prática, bastando apenas a criação de um novo tipo de iterador e associando-o ao grafo. O exemplo a seguir apresenta a utilização do iterador do tipo *Box* em um grafo, e posteriormente, o acesso a cada aresta que pertença à região definida pelo iterador.

```
//get current iterator
te::graph::AbstractIterator* oldIt =
  g->getIterator();

te::gm::Envelope* box =
  new te::gm::Envelope(...);

//set iterator
te::graph::BoxIterator* it =
  new te::graph::BoxIterator(g, box);

g->setIterator(it);

te::graph::Edge* edge = g->getFirstEdge();

// operation
while(edge)
{
  ...
  edge = g->getNextEdge();
}

g->setIterator(oldIt);

delete it;
```

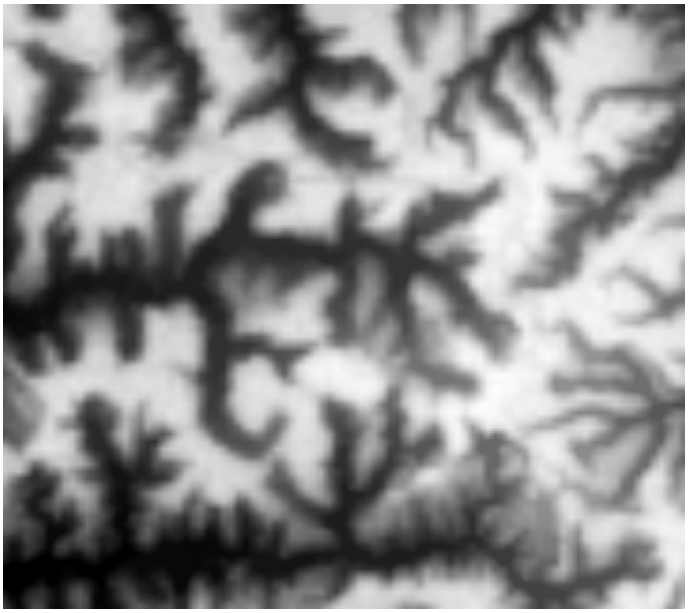
5.3 Extratores

A seguir são apresentados os grafos resultantes dos extratores implementados. Os resultados gerados servem para demonstrar a generalidade e robustez do modelo apresentado.

5.3.1 LDD

Como dito na seção 4.8.1 o LDD é uma matriz resultante de um processamento sobre um MNT. Um exemplo de utilização deste extrator é extração do grafo da Bacia Asu (Fig. 5.1).

(a)



(b)

Figura 5.1 - (a)Localização da Bacia Asu, (b) MNT da Bacia Asu.

A matriz que representa o LDD da Bacia Asu é uma grade regular com as seguintes dimensões:

- Linhas: 248
- Colunas: 280
- Resolução: 30m

O grafo resultante da extração a partir deste LDD é apresentado na Figura 5.2 e possui as seguintes quantidades de elementos:

- Vértices: 68.913
- Arestas: 68.227

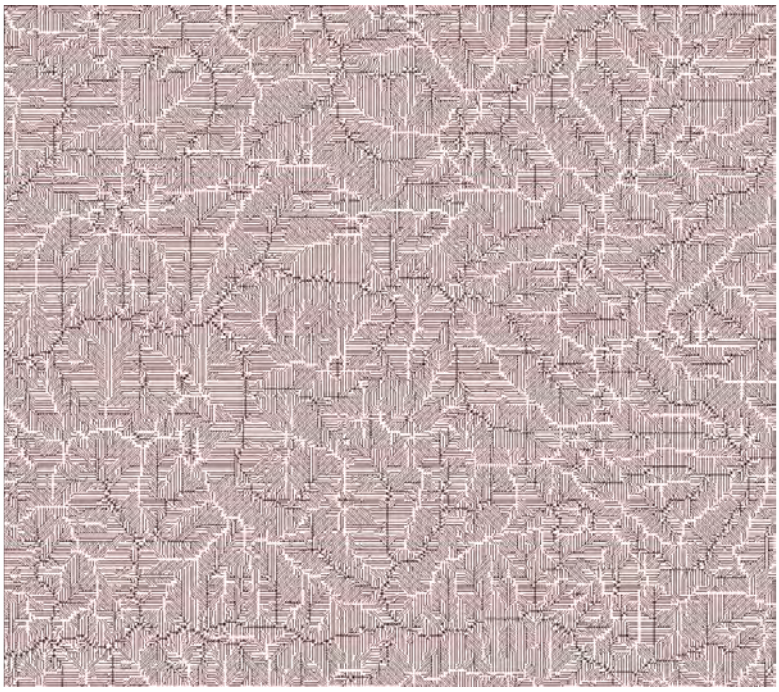


Figura 5.2 - Grafo resultante da extração a partir do LDD da Bacia Asu.

A seguir são apresentados algumas imagens em diferentes escalas deste grafo gerado, tendo como plano de fundo o MNT da Bacia Asu (Fig. 5.3).

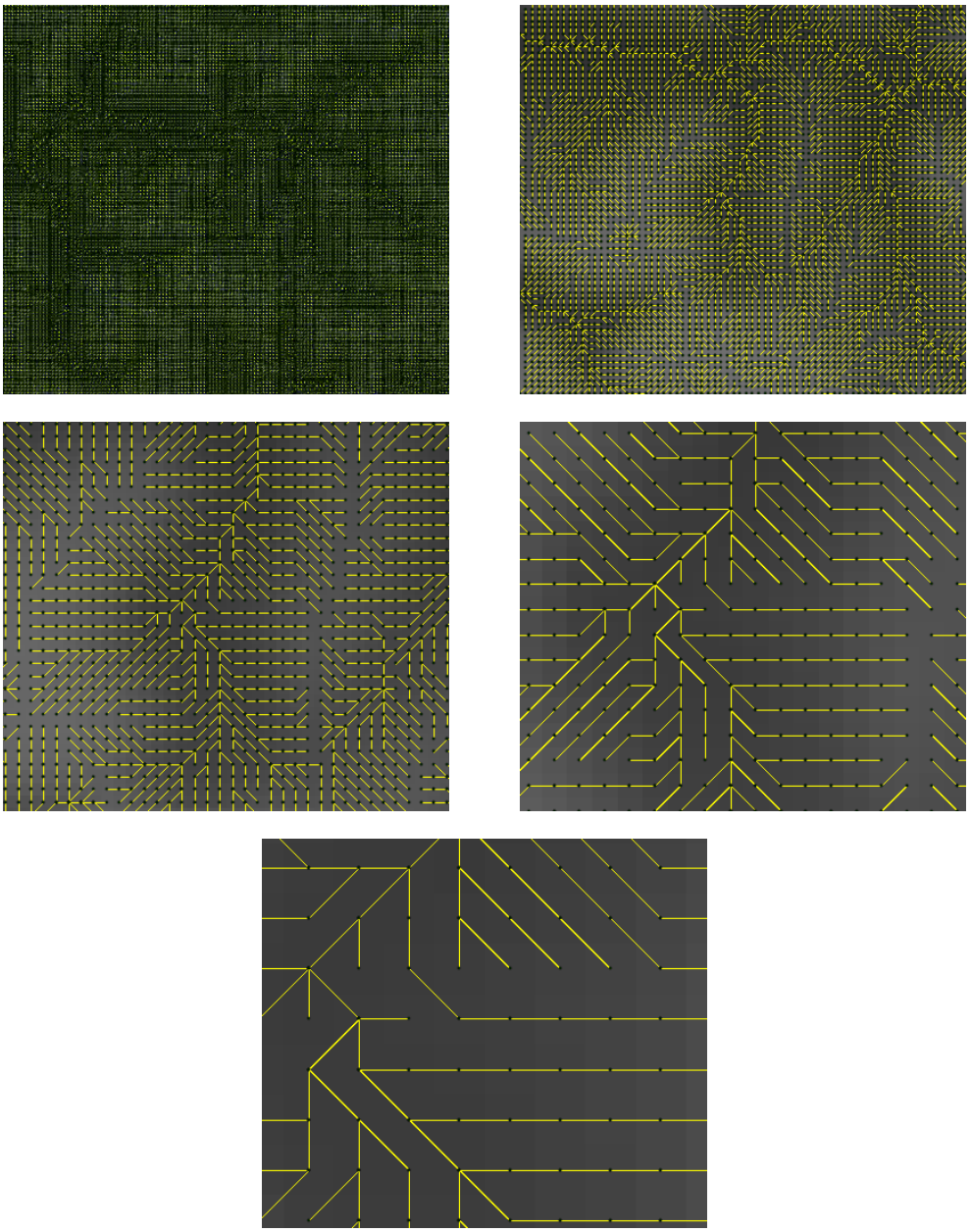


Figura 5.3 - Grafo da Bacia Asu em diferentes escalas de visualização.

5.3.2 Query

Uma outra forma de se gerar um grafo é aplicando uma restrição sobre algum grafo já existente, essa restrição é feita sobre algum atributo que esteja associado aos vértices ou arestas do grafo. Dessa maneira o extrator por *Query* funciona tendo um grafo existente e uma cláusula de restrição.

Neste exemplo foi utilizado o grafo gerado na seção anterior 5.3.1. Primeiramente

foi aplicada sobre o grafo uma função para o cálculo do fluxo acumulado para cada vértice (função que calcula o número de elementos antecessores conectados a um vértice). Com essa informação associada ao grafo é possível definir uma cláusula de restrição, por exemplo:

$$\text{Fluxo Acumulado} > 50$$

Aplicando essa restrição ao grafo da Bacia Asu, o grafo gerado (Fig 5.4) possui as seguintes quantidades de elementos:

- Vértices: 4.811
- Arestas: 4.776

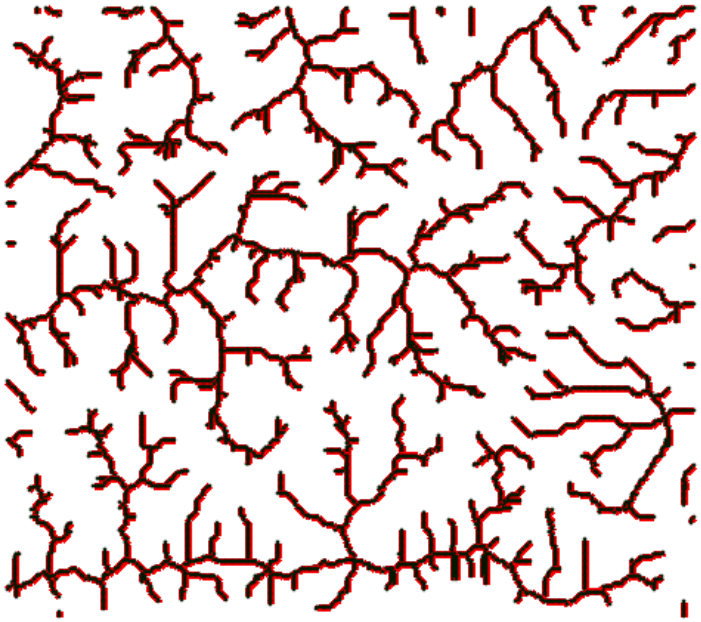


Figura 5.4 - Grafo resultante da extração por *Query* a partir do grafo da Bacia Asu.

A seguir são apresentados algumas imagens em diferentes escalas deste grafo gerado, tendo como plano de fundo o MNT da Bacia Asu (Fig. 5.5).

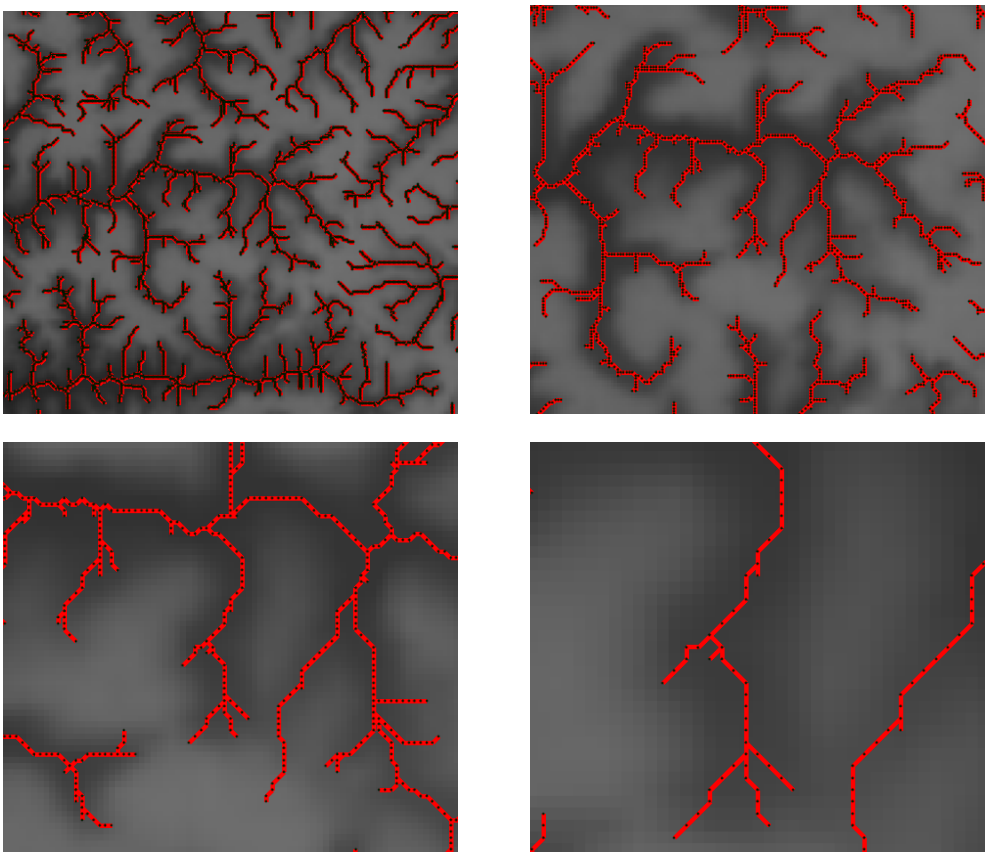


Figura 5.5 - Grafo resultante da restrição em diferentes escalas de visualização.

5.3.3 RAG

Um outro extrator utilizado foi o RAG, definido na seção 4.8.2. Este extrator possui como dado de entrada um conjunto de polígonos. Para este exemplo foram utilizados os municípios do Brasil como dado de entrada, possuindo 6.629 polígonos (Fig. 5.6).

Aplicando este extrator ao dado foi gerado um grafo contendo um vértice para cada município e uma aresta para cada polígono que se tocava (Fig 5.7), as seguintes quantidades de elementos foram geradas:

- Vértices: 5.512
- Arestas: 32.916

A seguir são apresentados algumas imagens em diferentes escalas deste grafo gerado, tendo como plano de fundo o dado vetorial com os municípios do Brasil (Fig. 5.8).

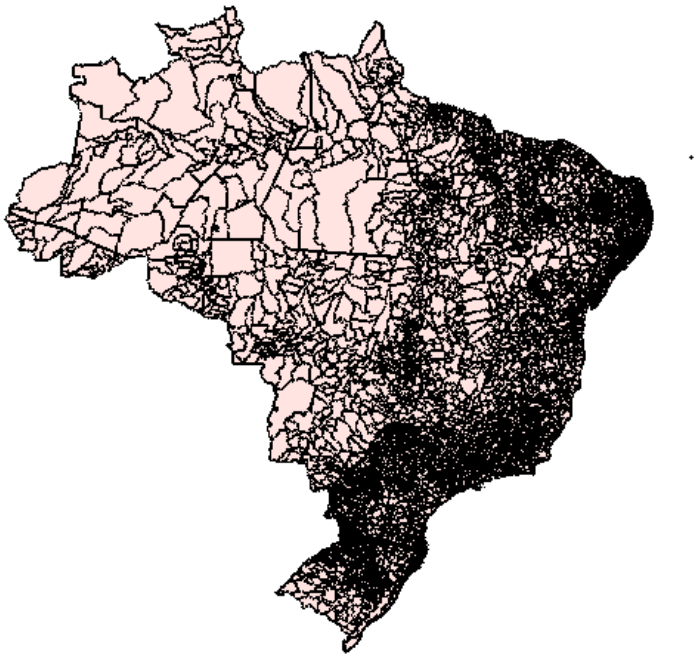


Figura 5.6 - Dado vetorial com os municípios do Brasil.

Figura 5.7 - Grafo resultante da extração por *RAG*.

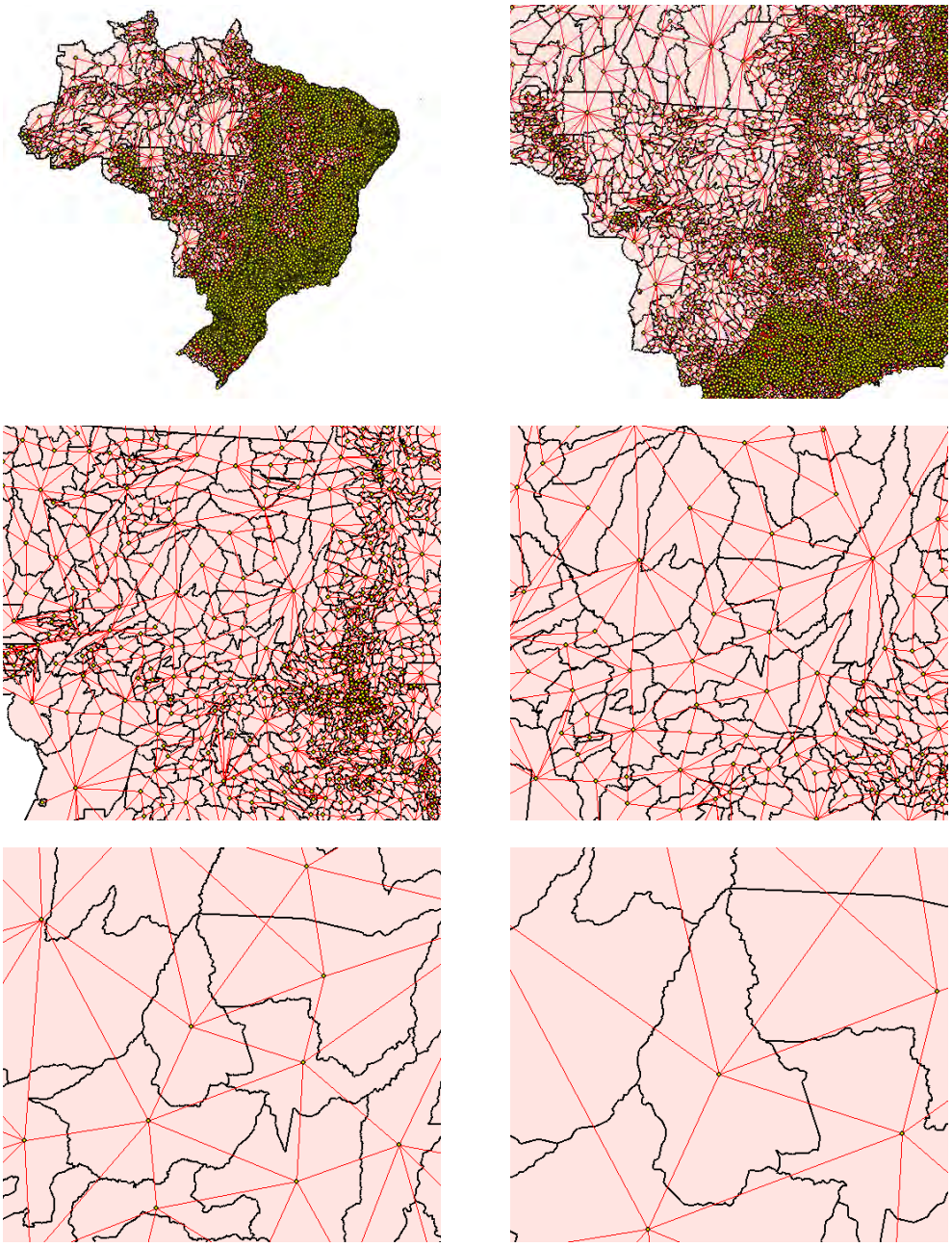


Figura 5.8 - Grafo resultante extração por RAG em diferentes escalas de visualização.

5.3.4 Flow

O último extrator implementado foi o extrator de fluxos, definido na seção 4.8.3. Este extrator utilizada como dados de entrada um dado vetorial e um dado tabular contendo as informações de fluxo. Para este exemplo foi utilizado o dado vetorial de municípios do Brasil e um dado tabular com informações de deslocamento de pacientes do SUS entre os municípios do Brasil.

O dado tabular utilizado contém 5.003 entradas e possui as seguintes colunas (Fig. 5.9):

- DE
- PARA
- FLUXO

	A	B	C
1	1100023	3505500	2
2	1100056	1100205	1
3	1100064	3505500	1
4	1100098	1100205	1
5	1100106	1100205	1
6	1100114	1100205	1
7	1100114	5208707	2
8	1100122	1100205	1
9	1100122	3505500	2
10	1100122	5208707	1
11	1100122	5103403	2
12	1100130	1100205	3
13	1100148	1100205	2
14	1100189	5103403	1
15	1100205	1100205	12
16	1100205	3505500	5
17	1100288	5208707	1
18	1100288	3505500	1
19	1100288	1100205	1
20	1100304	3505500	2
21	1100304	5103403	7

Figura 5.9 - Amostra do dado tabular utilizado no extrator de fluxos.

Os municípios do dado vetorial possuem um atributo de identificação igual aos utilizados no dado tabular para representar a origem e destino dos fluxos, com isso é possível associar a informação do dado tabular com os dados vetoriais.

Aplicando este extrator ao dado vetorial e ao dado tabular é gerado um grafo contendo um vértice para cada objeto do dado vetorial e uma aresta para cada linha da tabela. O grafo gerado por essa operação (Fig 5.10) possui as seguintes quantidades de elementos:

- Vértices: 5.512
- Arestas: 5.003

Figura 5.10 - Grafo gerado pelo extrator de fluxos.

A seguir são apresentados algumas imagens em diferentes escalas deste grafo gerado, tendo como plano de fundo o dado vetorial do contorno do Brasil (Fig. 5.11).

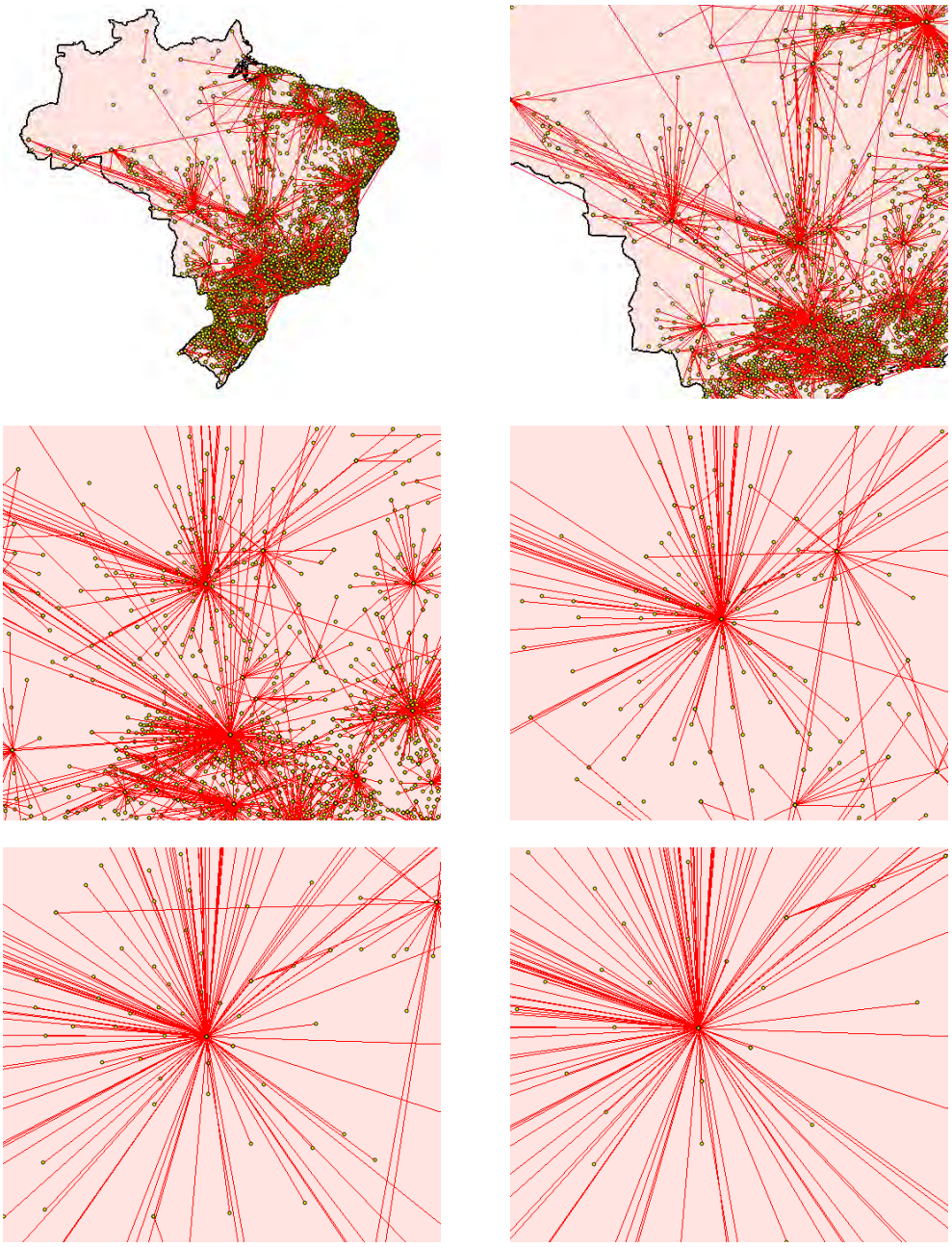


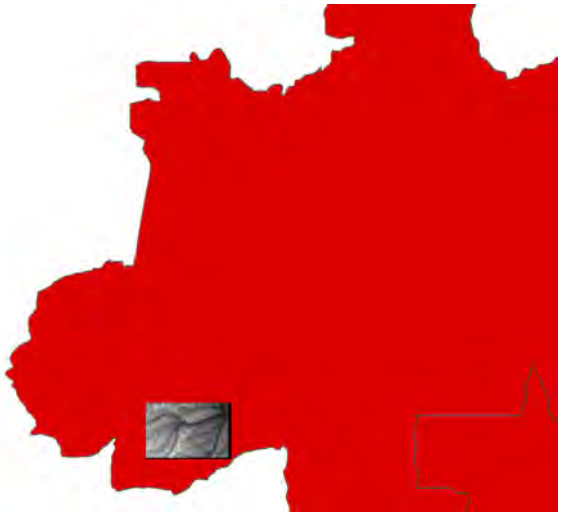
Figura 5.11 - Grafo resultante extração por fluxo em diferentes escalas de visualização.

5.4 Grandes Dados

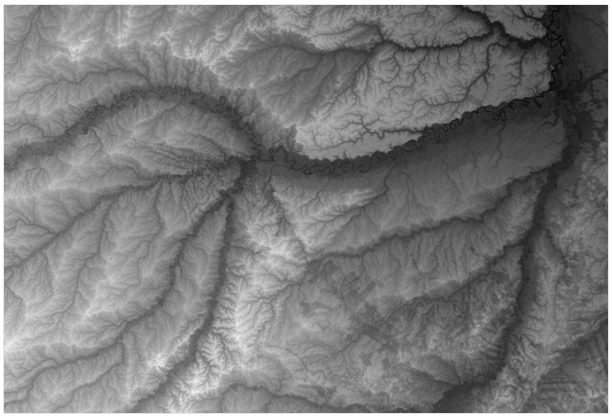
Um dos principais objetivos deste trabalho é a capacidade de tratar grandes quantidades de dados, possibilitando que grandes grafos possam ser gerados e tratados corretamente. O exemplo a seguir apresenta uma sequencia de operações desde a extração de um grafo até a detecção de sub-bacias.

O dado a ser utilizado nesse processamento é a Bacia Purus (Fig. 5.12), o MNT desta bacia possui as seguintes dimensões:

- Linhas: 1.598
- Colunas: 1.079
- Resolução: 90m



(a)



(b)

Figura 5.12 - (a)Localização da Bacia Purus, (b) MNT da Bacia Purus.

5.4.1 Extração do Grafo por LDD

Utilizando o extrator por LDD sobre o MNT da Bacia Purus, é gerado uma quantidade de vértices e arestas enorme, sendo necessário a utilização da estratégias de

cache para a realização desta operação.

Para este caso foram utilizadas as seguintes configurações de *cache*:

- Quantidade de pacotes em memória (*Cache Size*): 8
- Quantidade de elementos por pacote (*GraphData Size*): 200.000

O grafo gerado por essa extração possui as seguintes quantidades de elementos:

- Vértices: $\approx$ 6 milhões
- Arestas: $\approx$ 5.5 milhões

Uma pequena parte deste grafo extraido é mostrado na Figura 5.13.

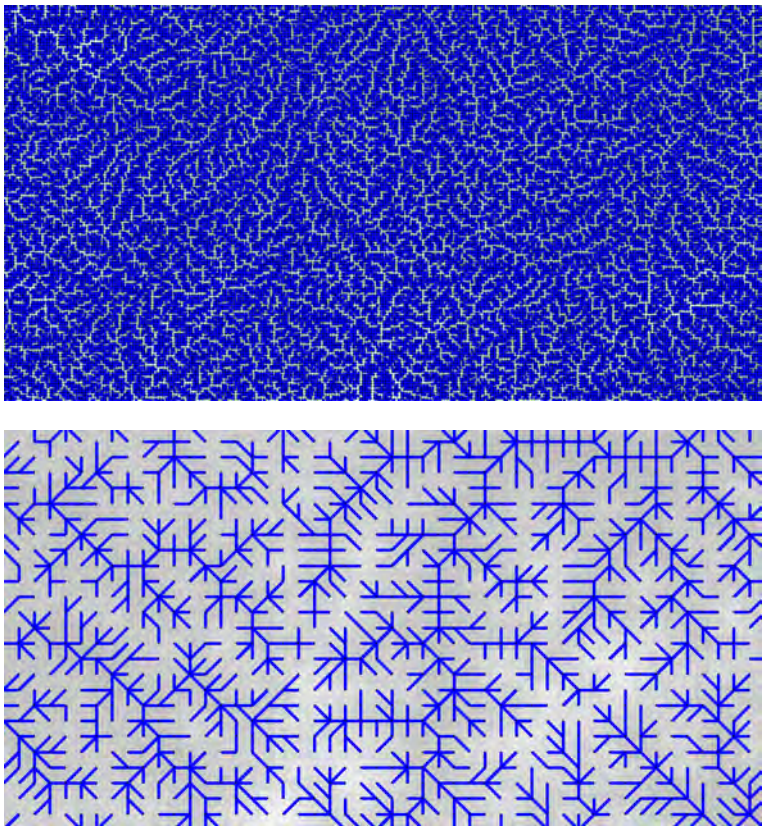


Figura 5.13 - Grafo resultante extração por LDD da Bacia Purus em duas escalas de visualização.

5.4.2 Extração do Grafo por *Query*

Uma operação interessante que pode ser feita sobre esse grafo gerado da Bacia Purus é a criação de um novo grafo baseado em uma restrição de altimetria. Associando aos vértices do grafo gerado na seção 5.4.1 a informação de altimetria disponível no próprio MNT da Bacia Purus, é possível criar uma restrição do tipo:

$$\text{Altimetria} < 125$$

O grafo gerado por essa restrição possui as seguintes quantidades de elementos (5.14):

- Vértices: 170.925
- Arestas: 161.343

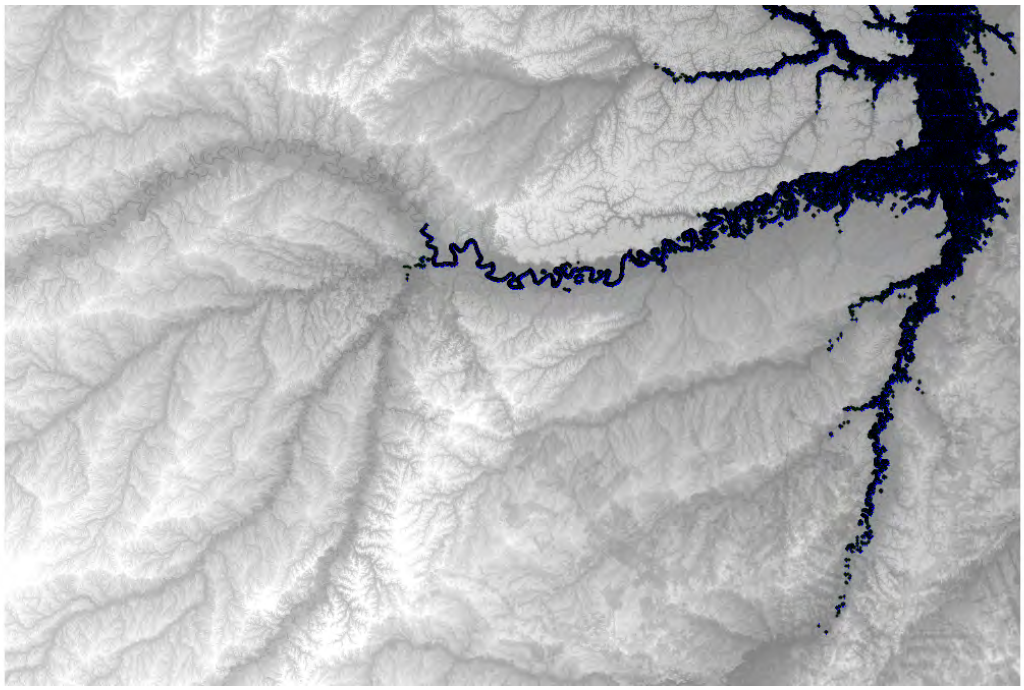
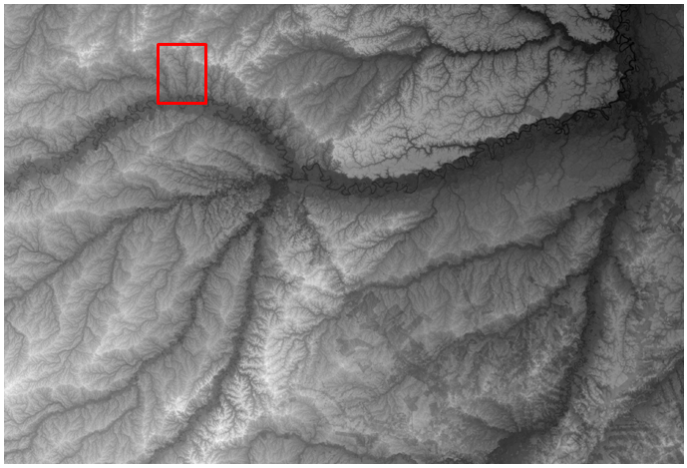


Figura 5.14 - Grafo da Bacia Purus com restrição de altimetria < 125m.

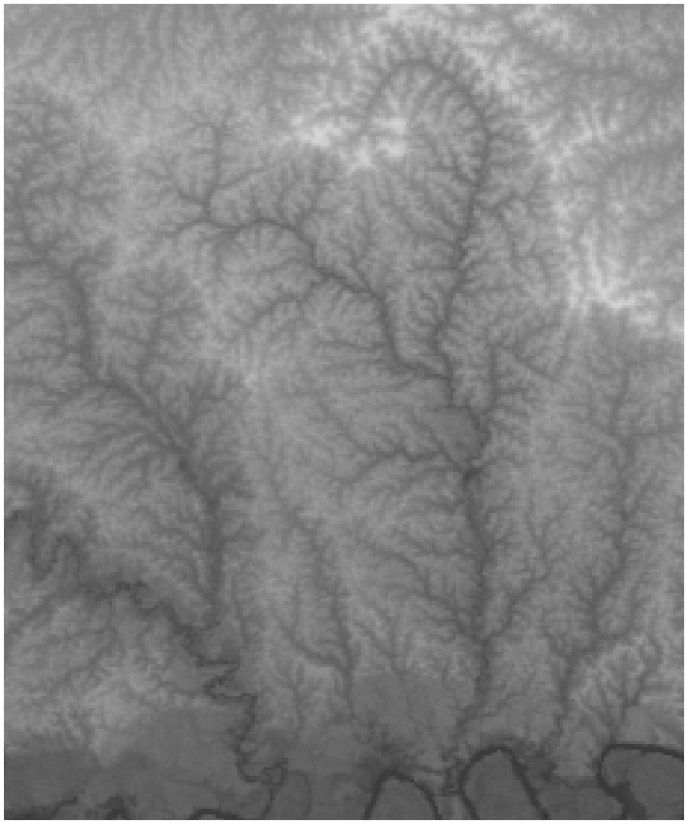
5.4.3 Detecção de Sub-bacias

Uma operação comum é a análise de apenas uma parte de um grafo. Dado um vértice é possível extrair um sub-grafo contendo todos os antecessores deste vértice definido. Seguindo esta lógica, dado um grafo que represente a rede hidrológica de uma região, como o grafo da Bacia Purus, é simples a extração de sub-bacias.

A Figura 5.15 apresenta uma região de interesse de onde será extraido uma sub-bacia.



(a)



(b)

Figura 5.15 - (a) Região de interesse da Bacia Purus representando uma sub-bacia, (b) Imagem da sub-bacia ampliada.

O grafo gerado a partir da detecção desta sub-bacia possui as seguintes quantidades de elementos (5.16):

- Vértices: 22.161
- Arestas: 22.160

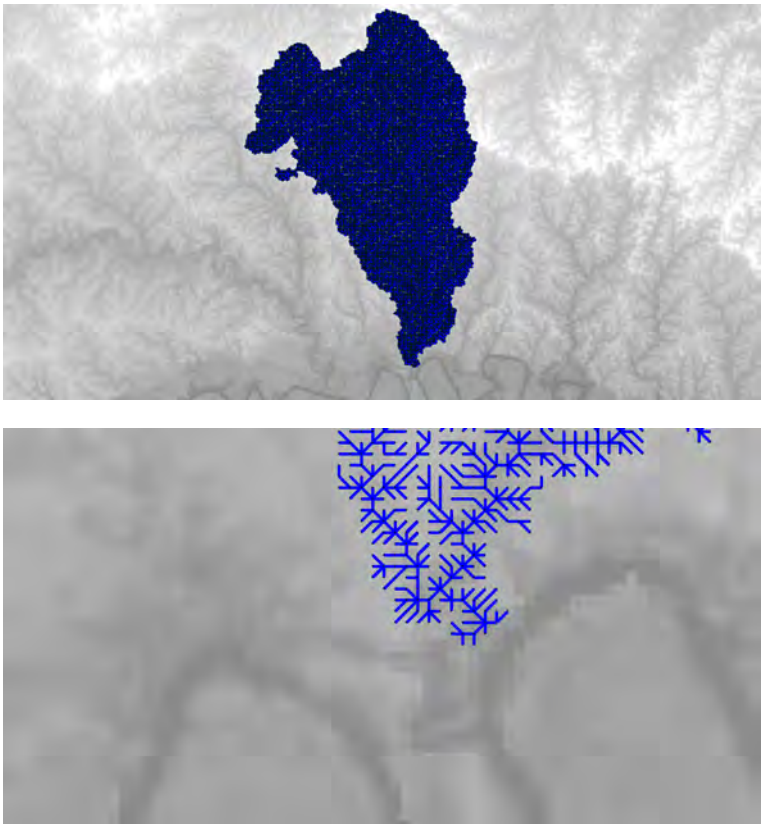


Figura 5.16 - Grafo de uma sub-bacia extraido do grafo da Bacia Purus em duas escalas de visualização.

5.4.4 Operações sobre Sub-bacias

Como ultimo exemplo de operações sobre grafos, pode-se fazer fazer uma análise sobre o grafo da sub-bacia gerado na seção 5.4.3. Sobre este grafo são realizadas as seguintes operações:

- cálculo fluxo acumulado;
- cláusula de restrição: fluxo acumulado > 150.

O grafo gerado a partir dessas operações é um grafo representando apenas o fluxo principal desta sub-bacia, contendo a seguinte quantidade de elementos (5.17):

- Vértices: 1.077
- Arestas: 1.075

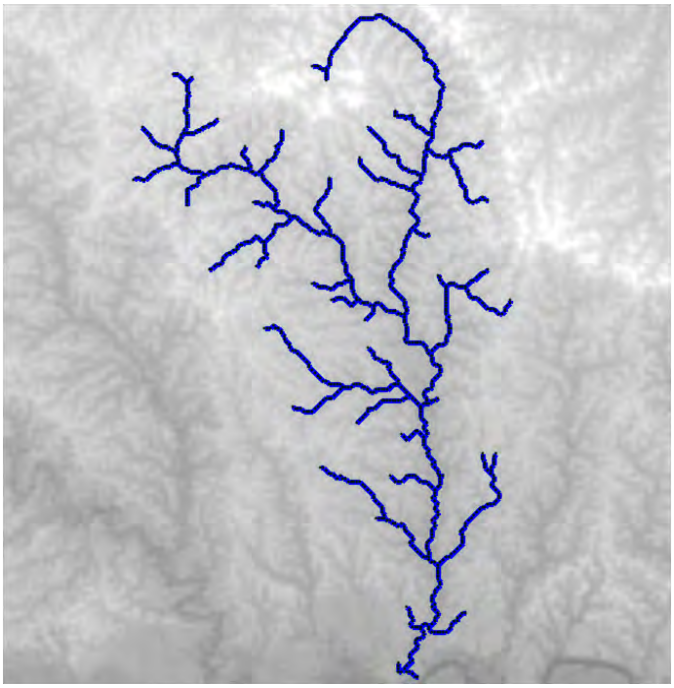


Figura 5.17 - Grafo da sub-bacia de Purus com restrição de fluxo acumulado > 150.

6 CONCLUSÕES

O tratamento de grafos demonstrou ser uma tarefa ampla e complexa, isso porque os grafos são utilizados como uma solução por uma gama muito grande de problemas e em cada solução uma abordagem específica é utilizada. Este fator explica as inúmeras aplicações e sistemas que abordam este assunto.

Neste sentido, este trabalho se baseou em um conjunto limitado de problemas para propor uma solução para armazenamento e manipulação de grafos em um contexto de geo-processamento. Foram definidos modelos e classes extensíveis que auxiliaram na flexibilidade do modelo.

O modelo proposto é válido e eficaz, permitindo que os objetivos fossem cumpridos. O principal problema abordado, que era a dificuldade em se trabalhar com grandes volumes de dados, também foi solucionado através da estratégia de *cache* apresentada.

Dentro deste contexto, este trabalho mostra-se a adequado para ser adotado como solução de projetos futuros na área de grafos, principalmente dentro do ambiente TerraLib. O resultado final deste trabalho é um *framework* para tratamento de grafos com as seguintes funcionalidades:

- Definição de um modelo abstrato de grafo;
- Persistência do grafo em fontes de dados;
- Estratégias de recuperação e *cache* dos dados;
- Definição de iteradores para se percorrer os dados;

Motivado pelo grande interesse da comunidade científica no estudo de grafos, espero que este trabalho contribua de forma significativa para os projetos de Sistemas de Informações Geográficas na área de grafos e redes do INPE. O desenvolvimento utilizando o ambiente TerraLib contribui para o crescimento da biblioteca e para a multiplicação do conhecimento, visto o caráter livre e código aberto do projeto.

6.1 Trabalhos Futuros

Este trabalho poderá ser posteriormente estendido, implementando alguns dos tópicos apresentados a seguir:

- utilização de uma biblioteca de terceiros para prover os algoritmos de percorrimento e processamento sobre grafos. Essa deverá ser uma tarefa relativamente simples devido ao nível de abstração de grafo definida neste trabalho. A biblioteca sugerida para tal acoplamento é a BGL (SIEK et al., 2002).

- criar um esquema de paralelização para a busca dos elementos no repositório. Uma vez que não temos mais a limitação da memória para a manipulação de dados, o gargalo passa a ser a velocidade de acesso aos dados no repositório;

- comparação com outras fontes de dados. Neste trabalho foi definida uma forma inicial de armazenamento dos dados em bancos de dados relacionais, é interessante estudar a possibilidade de esses dados estarem armazenados em outras fontes de dados, como por exemplo, os bancos de dados específicos para armazenamento de grafos, como foi analisado neste trabalho.

REFERÊNCIAS BIBLIOGRÁFICAS

ALBERTON, L. **RDBMS in the social networks age**: Ctes and window functions. London, UK: [s.n.], 2010. Disponível em: <http://www.slideshare.net/quipo/rdbms-in-the-social-networks-age>. 15

BOLLOBÁS, B. **Modern graph theory**. New York, NY, USA: Springer-Verlag, 1998. (Graduate Texts in Mathematics). 5

BONDY, A.; MURTY, U. **Graph theory**. New York: Springer, 2008. (Graduate Texts in Mathematics). 5, 6

CAMARA, G.; VINHAS, L.; QUEIROZ, G. R.; FERREIRA, K. R.; MONTEIRO, A. M.; CARVALHO, M.; CASANOVA, M. Terralib: an open-source gis library for large-scale environmental and sócio-economic applications. In: **Open Source Approaches to Spatial Data Handling**. Berlin: Springer-Verlag, 2008. 13

CASANOVA, M. A.; CÂMARA, G.; JR., C. D.; VINHAS, L.; QUEIROZ, G. R. d. **Bancos de dados geográficos**. São José dos Campos: Mundogeo, 2005. Disponível em CD-ROM na Biblioteca INPE-12830-PRE/8120. 9

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Introduction to algorithms**. Massachusetts - USA: The MIT Press, 2001. 8

DEITEL, P. J.; DEITEL, H. **C++ como programar**. 3th. ed. Porto Alegre: Bookman, 2002. 45

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design patterns**: elements of reusable object-oriented software. 1. ed. Indianapolis, US: Pearson Education, 1994. 45, 51

GRAPH-DATABASE.ORG. **Graph database**. 2011. Disponível em: <http://www.graph-database.org>. Acesso em: 2012-08-28. 12

GUTTMAN, A. R-trees: a dynamic index structure for spatial searching. In: **Proceedings...** New York, NY, USA: ACM, 1984. (SIGMOD '84). 10

HARARY, F. **Graph theory**. USA: Addison-Wesley, 1969. (Addison-Wesley Series in Mathematics). 5

HIMSOLT, M. **GML**: a portable graph file format. 94030 Passau, Germany: [s.n.], 1999. Disponível em: <http://www.infosun.fim.uni-passau.de/Graphlet/GML/gml-tr.html>. 3

KASTL, D.; JUNOD, F. Foss4g routing with pgrouting tools, openstreetmap road data and geoext. In: **FOSS4G**. Barcelona: [s.n.], 2010. 19, 20

KORTING, T. S. **Um paradigma para re-segmentação de imagens de alta resolução**. Dissertação (Mestrado) — Instituto Nacional de Pesquisas Espaciais (INPE), São José dos Campos, 2007. 27, 50

MEHLHORN, K.; NäHER, S. Leda: a platform for combinatorial and geometric computing. ACM, New York, NY, USA, 1995. 1

MURRAY, C. **Oracle Spatial**: topology and network data models developer's guide. Redwood City, CA, USA: [s.n.], 2009. 16, 18, 19, 20

OGC. **Open Geospatial Consortium (OGC)**. 2011. Disponível em: <http://www.opengeospatial.org/>. Acesso em: 13 fevereiro 2011. 11, 13

OLIVEIRA, E. X. G. de. **A multiplicidade do único território do SUS**. Tese (Doutorado) — Escola Nacional de Saúde Pública, Rio de Janeiro, 2005. 27, 50, 51

QUEIROZ, G. R.; FERREIRA, K. R.; VINHAS, L.; CÂMARA, G.; MONTEIRO, A. M. V.; GARRIDO, J. C. P.; HARA, L.; XAVIER, M.; CASTEJON, E. F.; SOUZA, R. C. M. d. Terralib 5.0: supporting data-intensive giscience. In: WORKSHOP DOS CURSOS DE COMPUTAÇÃO APLICADA DO INPE, 10. (WORCAP), São José dos Campos. **Anais...** São José dos Campos: Instituto Nacional de Pesquisas Espaciais (INPE), 2010. 13, 14

ROSIM, S. **Estrutura baseada em grafos para representação unificada de fluxos locais para modelagem hidrológica distribuída**. Tese (Doutorado) — Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2008. 2, 3, 49

ROSIM, S.; MONTEIRO, A. M. V.; RENNÓ, C. D.; SOUZA, R. C. M. d.; SOARES, J. V. Terrahidro: uma plataforma computacional para o desenvolvimento de aplicativos para a análise integrada de recursos hídricos. In: SIMPÓSIO BRASILEIRO DE SENSORIAMENTO REMOTO, 11. (SBSR), Belo Horizonte. **Anais...** São José dos Campos: INPE, 2003. p. 2589 – 2596. 2, 27

RYDEN, K. Opengis implementation specification for geographic information: simple feature access - part 1:common architecture. **Version 1.1.0, OGC**

05-126., 2005. Disponível em:
<http://www.opengeospatial.org/standards/sfa>. 11

SIEK, J. G.; LEE, L.-Q.; LUMSDAINE, A. **The Boost Graph Library**: user guide and reference manual. Boston, MA, USA: [s.n.], 2002. 1, 76

TEAM, N. **The Neo4j manual v1.8.M06**. Neo Technology, 2012. Disponível em: <http://www.neotechnology.com>. 22, 23

TECHNOLOGIES, S. **Dex 4.6.0**: technical documentation. Barcelona: [s.n.], 2011. Disponível em: <http://www.sparsity-technologies.com/dex_tutorials>. Acesso em: 2011-09-28. 24

_____. **Dex starting guide**. Barcelona: [s.n.], 2011. Disponível em: <http://www.sparsity-technologies.com/dex_tutorials>. Acesso em: 2011-09-21. 24, 25

Anexo A

Paper submetido ao GEOINFO 2012

Armazenamento e Processamento de Grandes Grafos em Bancos de Dados Geográficos

Eric S. Abreu[1], **Sergio Rosim**[1], **João Ricardo de F. Oliveira**[1], **Gilberto Ribeiro**[1], **Luciano V. Dutra**[1]

[1]Instituto Nacional de Pesquisas Espaciais (INPE)
Caixa Postal 1758 – 12227-010 – São José dos Campos – SP– Brasil

{eric,sergio,joao,gribeiro,luciano}@dpi.inpe.br

Abstract. *This paper presents a methodology that allows the storage and manipulation of large graphs in geographic database by defining a set of relational tables. These tables represent the graph by using the connection information and attributes, as well as its metadata. The graphs discussed in this work are those that can be spatially defined. A library of geo-processing and a cache policy are also used to optimize the access to this data.*

Resumo. *Este trabalho apresenta uma metodologia que permite o armazenamento e manipulação de grandes grafos em banco de dados geográficos através da definição de um conjunto de tabelas relacionais. Essas tabelas representam o grafo através das informações de conexão e atributos, bem como seus metadados. Os grafos abordados nesse trabalho são aqueles que podem ser espacialmente definidos. Uma biblioteca de geo-processamento e uma política de cache também são utilizadas para otimizar o acesso a esses dados.*

1. Introdução

Utilizamos a estrutura de grafos quando queremos representar a conectividade e a relação entre objetos de um determinado conjunto. Matematicamente os grafos são definidos por um par ordenado *(V, A)*, onde *V* é um conjunto de vértices e *A* uma relação binária sobre *V*, cujos elementos são denominados de arestas [Bollobás 1998].

No caso específico deste trabalho estamos interessados em grafos que sejam espacialmente definidos, ou seja, que seus vértices possuam localizações espaciais bem definidas, tais como redes de infra-estrutura (transporte, saneamento e energia elétrica). Ao armazenarmos esse tipo de informação em banco de dados geográficos, nos é permitido uma série de operações que facilitam o processamento desses dados. Atualmente existem bancos de dados relacionais dedicados ao armazenamento de grafos (Oracle Spatial Network Data Model [Murray 2009], PgRouting [Kastl and Junod 2010], etc) ou

mesmo bancos de dados não relacionais que são específicos para o armazenamento de grafos (Neo4J [Team 2012], DEX [Martínez-Bazan et al. 2007], etc).

A proposta deste trabalho é criar um pacote para manipulação de grafos em bibliotecas geográficas, tendo como principal abordagem a definição de um modelo de grafos genérico para armazenamento em banco de dados relacionais que não seja dependente de um SGBD (Sistema Gerenciador de Banco de Dados), bem como a criação de funções de recuperação e persistência que sejam capazes de manipular grandes quantidades de dados sem perda de desempenho. Para a implementação desse projeto escolhemos a biblioteca geográfica *Terralib5* [Queiroz et al. 2011]

A seção 2 descreve a metodologia desenvolvida, a seção 3 exemplifica uma estratégia de extração de grafos; e por fim a seção 4 apresenta as conclusões.

2. Metodologia

Esta seção apresenta de que maneira uma estrutura de grafo pode ser armazenada em um SGBD e como recuperar essas informações de forma eficiente através do uso de uma biblioteca de geo-processamento, mantendo a robustez do processamento independente do tamanho do grafo.

2.1. Modelo de Persistência

Para que um grafo possa ser armazenado em um SGBD e posteriormente recuperado é necessário um conjunto de metadados que o descrevam. Informações que indiquem quais atributos estão sendo associados aos grafos, qual tabela está sendo utilizada para armazenar os dados do grafo, são de fundamental importância para sua utilização. A Figura 1 e a Tabela 1 exemplificam este modelo.

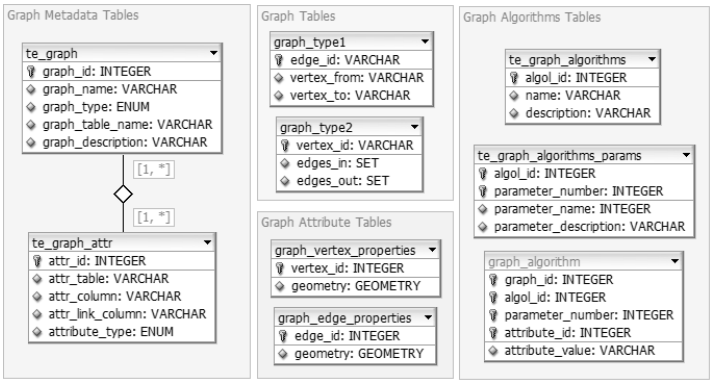


Figura 1. Modelo de Persistência.

2.2. Modelo de Dados

O modelo de dados utilizado para representar os elementos do grafo usa o conceito de classes. É possível definir um modelo flexível através da definição abstrata das princi-

Tabelas	
te_graph	Metadados de um grafo.
te_graph_attr	Metadados dos atributos de um grafo.
graph_type1	Tabela de dados ordenado por arestas.
graph_type2	Tabela de dados ordenado por vértices.
graph_vertex_properties	Propriedades dos vértices.
graph_edge_properties	Propriedade das arestas.
te_graph_algorithms	Metadado dos algoritmos.
te_graph_algorithms_params	Metadado dos parâmetros dos algoritmos.
graph_algorithms	Lista dos grafos associados a algoritmos.

Tabela 1. Tabelas do Modelo de Persistência

pais operações; funções de inserção, remoção e acesso aos elementos são definidas em uma classe virtual chamada de *AbstractGraph*, facilitando extensões para tipos de grafos específicos (como bidirecionais, unidirecionais e etc.). A Figura 2 ilustra o modelo de dados.

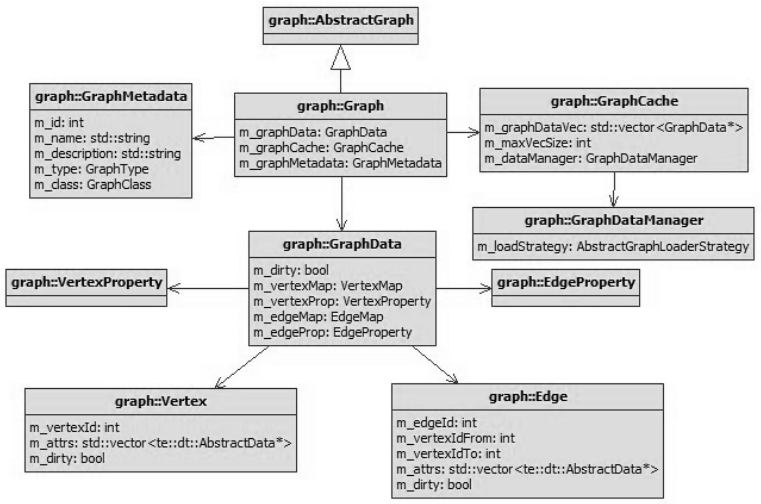


Figura 2. Modelo de Dados.

Uma breve descrição das classes é feita abaixo:

- *Graph* - Uma implementação concreta da classe AbstractGraph.
- *GraphData* - Representação de um pacote de dados.
- *GraphDataManager* - Acesso aos dados diretamente no repositório.
- *GraphCache* - Representação de uma estrutura de cache.
- *Vertex* - Representação do objeto vértice.
- *Edge* - Representação do objeto aresta.
- *VertexProperty* - Lista dos metadados dos atributos associado aos vértices.
- *EdgeProperty* - Lista dos metadados dos atributos associados às arestas.

2.3. Acesso aos Dados

A classe *GraphDataManager* é considerada uma ponte para acesso aos dados do grafo e fornece métodos que retornam pacotes *(GraphData)* dado um identificador de um elemento, seja vértice ou aresta. Esta classe possui um atributo que é a classe abstrata *AbstractGraphLoaderStrategy*.

Foi definido o conceito de *LoaderStrategy* que determina a maneira que o dado deve ser carregado. Três estratégias foram consideradas, como mostra a Figura 3.

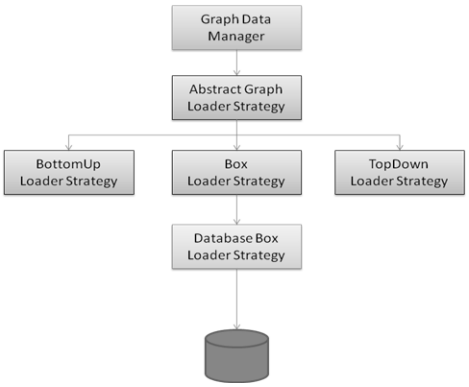


Figura 3. Estratégias de carga dos dados.

- *BottomUp*: carrega um conjunto de objetos a partir do objeto procurado, percorrendo o grafo de forma reversa (grafo direcionado).
- *TopDown*: carrega um conjunto de objetos a partir do objeto procurado, percorrendo o grafo seguindo seu fluxo normal (grafo direcionado).
- *Box*: carrega um conjunto de objetos tendo como objeto central o item procurado.

Neste trabalho é feito o armazenamento e acesso aos dados em um banco de dados relacional, utilizando o *Box* como estratégia de carga dos dados.

2.4. Cache

Uma parte importante deste projeto e que irá auxiliar no desempenho de acesso aos elementos do grafo é a estrutura de *cache*. A classe *GraphCache* é constituída por um vetor de *GraphData* e possui um atributo que é a classe *GraphDataManager*, utilizado quando um elemento desejado não está presente no *cache* (Figura 4).

Duas observações importantes a fazer a respeito dessa estrutura de *cache* são:

- Tamanho do vetor: quanto mais pacotes o cache possuir, maiores são as chances de o elemento procurado estar carregado, porém em cada busca ele terá que pesquisar em mais pacotes para ver se o elemento está carregado.

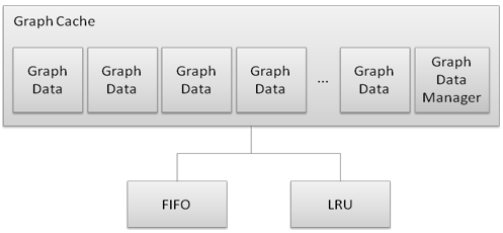


Figura 4. Modelo de Cache.

- Política de *cache*: o tamanho do vetor de armazenamento dos pacotes é configurável, mas uma vez atingido o limite máximo é necessário começar a descartar pacotes. Foram definidas duas políticas de *cache*: *FIFO (First In First Out)* e *LRU (Least Recently Used)*.

Essa política passa a ser simples devido ao fato de os elementos estarem agrupado em pacotes. O controle é feito por pacotes e não por elementos individuais, evitando uma sobrecarga de checagens e controles. Em testes realizados, a estratégia de bloco único é 9%, em média, mais rápido do que a estratégia por múltiplos blocos. Entretanto, ao utilizar um bloco único, as políticas de *cache* se tornam mais complexas, sendo necessário um controle individual de cada elemento para verificar seus acessos.

Outro fator importante dessa estrutura é a paralelização: se usarmos o acesso ao cache de forma seqüencial, toda a aplicação irá ficar parada esperando que um novo elemento seja carregado, criando um grande gargalo no processamento.

3. Extração dos Grafos

Para comprovar a real capacidade de armazenamento do grafo no SGBD e posteriormente sua recuperação, foi implementada uma metodologia de extração de grafos a partir de um *MNT (Modelo Numérico de Terreno)*, seguindo a especificação definida em [Rosim 2008].

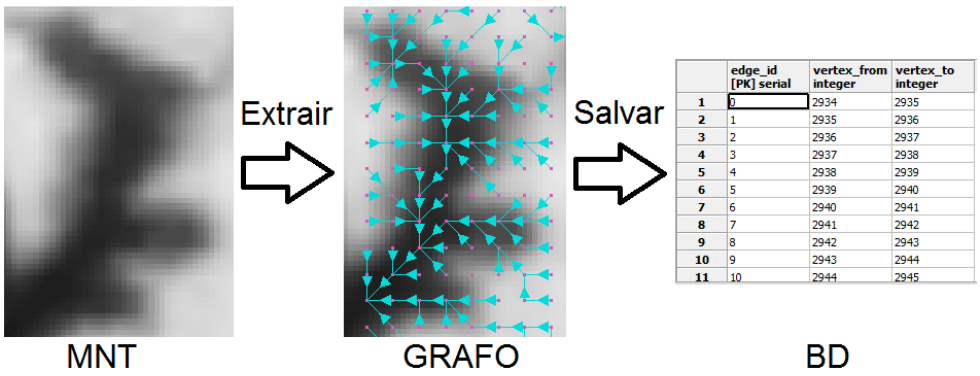


Figura 5. Extração de grafos a partir de um MNT.

Em testes realizados, os grafos foram corretamente extraídos e armazenados no banco dados.

4. Conclusão

Percebemos que o modelo adotado é eficiente, pois permite que aplicações distintas, como modelagem hidrológica [Rosim 2008] e matrizes de aproximação [Aguiar et al. 2003] (ambas feitas usando a TerraLib) possam utilizar o mesmo arcabouço, independente do tamanho dos dados a ser processado. O modelo de tabelas para armazenar o grafo que tem demonstrado melhores resultados é o ordenado por arestas. Mesmo considerando que o banco de dados relacional não tem a mesma eficiência que um banco de dados para grafos, vale ressaltar que o modelo se adapta muito bem a bancos de dados espaciais, o que facilita sua utilização junto à *TerraLib*.

Referências

Aguiar, A. P. D., Câmara, G., Monteiro, A. M. V., Cartaxo, R., and de Souza, M. (2003). Modelling spatial relations by generalized proximity matrices. In *V Brazilian Symposium in Geoinformatics - GeoInfo*.

Bollobás, B. (1998). *Modern graph theory*. Springer-Verlag.

Kastl, D. and Junod, F. (2010). W-10: Foss4g routing with pgrouting tools, openstreetmap road data and geoext. In *FOSS4G*.

Martínez-Bazan, N., Muntés-Mulero, V., Gómez-Villamor, S., Nin, J., Sánchez-Martínez, M.-A., and Larriba-Pey, J.-L. (2007). Dex: high-performance exploration on large graphs for information retrieval. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*.

Murray, C. (2009). *Oracle Spatial Topology and Network Data Model Developers Guide*. Oracle.

Queiroz, G. R., Ferreira, K. R., Vinhas, L., Câmara, G., Monteiro, A. M. V., Garrido, J. P., Xavier, L. H. M., Castejon, E. F., and Souza, R. C. M. (2011). Terralib5.0: Supporting data-intensive giscience. In *X Worcap - Instituto Nacional de Pesquisas Espaciais - INPE*.

Rosim, S. (2008). *Estrutura baseada em grafos para representação unificada de fluxos locais para modelagem hidrológica distribuída*. PhD thesis, Instituto Nacional de Pesquisas Espaciais.

Team, T. N. (2012). *The Neo4j Manual v1.8.M06*. Neo Technology.