- Relatório Técnico -

Este é o primeiro relatório do projeto **TH-APP** e representa as atividades executadas durante o primeiro mês.

As atividades executadas foram:

- Estudo do que são Áreas de Preservação Permanente – APP's,
- Análise da biblioteca TerraLib5,
- Estudo e implementação de uma estrutura de grafos.

APP

Áreas de Preservação Permanente – APP's, segundo o novo código florestal, podem ser definidos como uma área protegida, coberta ou não por vegetação nativa, com a função ambiental de preservar os recursos hídricos, a paisagem, a estabilidade geológica, a biodiversidade, facilitar o fluxo gênico de fauna e flora, proteger o solo e assegurar o bem-estar da população humana.

Atualmente as metodologias e processos utilizados para a definição de APP's são basicamente manuais e dependem do conhecimento de poucos especialistas e de interpretações particularizadas da lei.

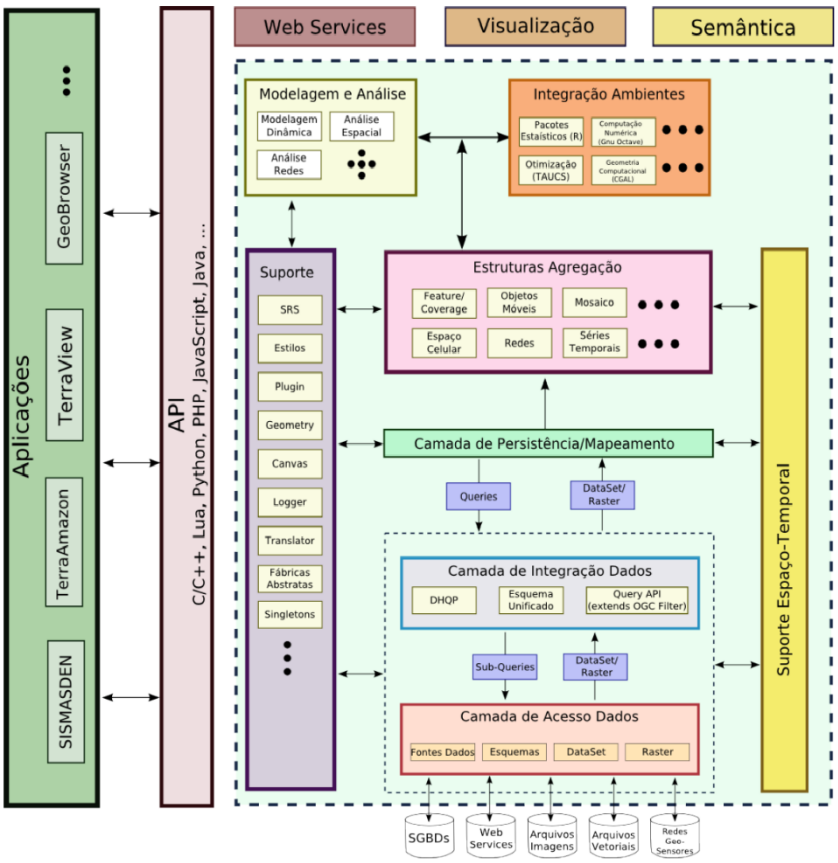
Alguns fatores configuram uma área como APP, dentre esses fatores, alguns são identificados através de dados secundários ou de dados de sensoriamento remoto. Outros são identificados através de operações cartográficas para serem delimitados.

Este projeto tem como foco a utilização de estruturas de grafos para automatizar a delimitação destas áreas.

TerraLib

A **TerraLib** é um projeto que visa atender grandes demandas institucionais na área de Geoinformática, criando um ambiente para pesquisa e desenvolvimento de inovações em geoprocessamento. A **TerraLib 5** é uma plataforma de software tendo seu núcleo desenvolvido em C++ e tem como principais características:

- **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 para diferentes finalidades;
- **Estruturas de agregação**: fornece um mecanismo extensível capaz de introduzir novas representações de alto nível.

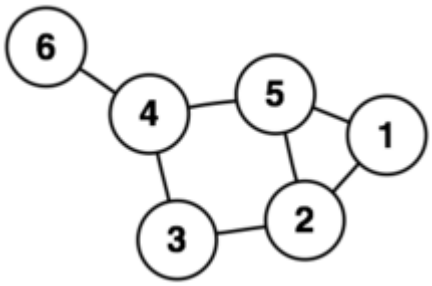


Baseado nessa última característica, que nos permite criar novos tipos de representação, e utilizando das facilidades de acesso aos dados, é que iremos criar um **framework** para grafos. Podemos ver o modelo de grafo proposto neste trabalho como sendo uma estrutura de agregação da **TerraLib**.

Grafos

Grafo é uma estrutura constituída por um conjunto de vértices e um conjunto de arestas que ligam pares de vértices, como pode ser visto na figura abaixo.

- Os vértices podem ser utilizados para representar cidades, pessoas, números, etc.; na figura, os vértices são os elementos (1,2,3,4,5,6).
- As arestas representam a existência de ligações entre os vértices, valor de ligação, distância entre nós, etc.



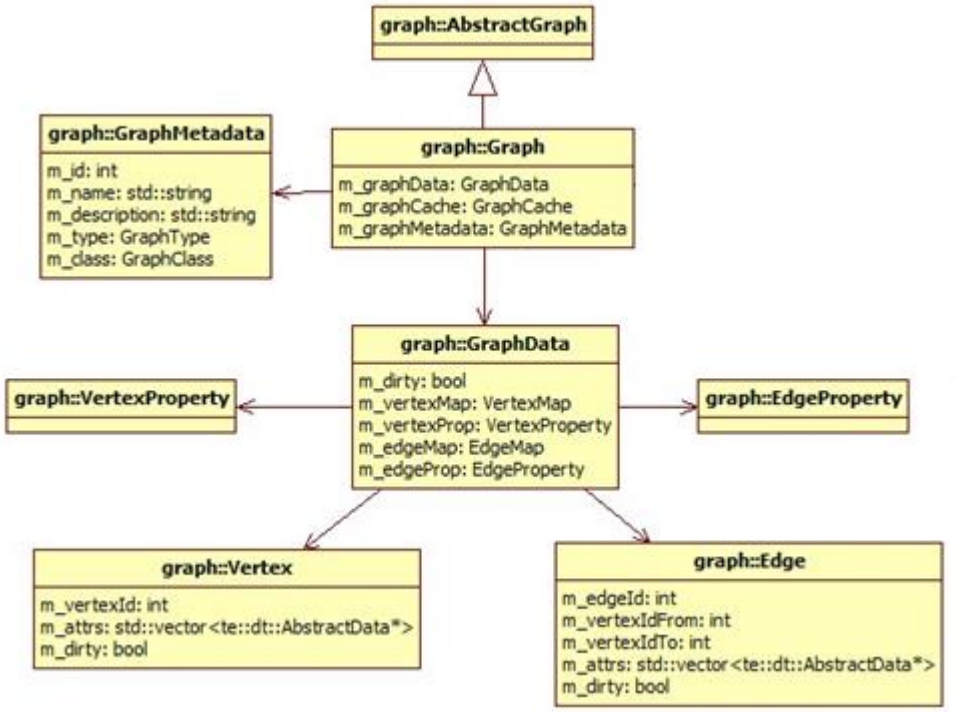
Neste projeto o grafo será utilizado para representar redes de drenagem. A representação de rios na forma estruturada de grafos visa não somente a desenvolvimentos deste projeto, como também a inúmeros trabalhos ligados ao meio ambiente, do qual as APP's representam uma parte importante.

Grafos – 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:



- *AbstractGraph*: definição da interface abstrata de um grafo genérico;
- *Graph*: implementação concreta de um grafo;
- *GraphMetadata*: conjunto de atributos que descrevem o grafo;
- *Vertex*: objeto que representa um vértice;
- *VertexProperty*: indica 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);

Grafos – Interface e Manipulação

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 podem ser utilizados.

Create

O primeiro exemplo mostra como criar um objeto do tipo grafo. Para que isso seja possível, é necessário tomar algumas decisões e informar alguns parâmetros, tais como:

- Onde o grafo será armazenado;
- Como o dado será armazenado;
- Qual a forma de carregar os dados armazenados;

Todas essas informações são definidas em um conjunto de parâmetros. A partir dessas informações, o **framework** saberá criar corretamente o tipo de grafo

```
// 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_CACHE_POLICY"] = "FIFO";

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

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 associado a 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**.

Iterators

A utilização de iteradores é muito prática, bastando apenas a criação de um novo tipo de iterador e sua associação 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;
```