# Documentação do Projeto Final:
# Malhas Aéreas em Grafos

Guilherme Diniz - 637423
Leonardo Antunes - 637295
Tulio Soares - 622281

## Introdução

Neste projeto final da matéria de *Algoritmos Computacionais em Grafos* lecionado pela professora *Eveline Veloso*, nos foi requisitado o desenvolvimento de uma aplicação que permita a representação e utilização de uma malha aérea. Cada malha aérea contém os dados sobre os aeroportos da região representada, além das rotas e voos disponíveis entre eles.

Como é esperado de um trabalho final de matéria, o enunciado apresentou diversas exigências quanto a modelagem do projeto, além de diversos desafios que nós devíamos encontrar soluções dentro dos assuntos abordados em aula. Em termos simples, as exigências foram:

- A malha aérea em questão deve apresentar uma estrutura de dados capaz de suportar tanto suas rotas quanto seus voos.
- Para representar as rotas e voos, devem ser criados um grafo não dirigido e um grafo dirigido, respectivamente. Os vértices de cada grafo deve representar aeroportos e as arestas devem representar os voos ou rotas entre eles.
- Cada rota e voo apresenta a distância entre os aeroportos em quilômetros e a duração de voo entre os aeroportos.
- Cada voo deve apresentar, também, os horários de partida deste.
- O projeto deve suportar a leitura de um arquivo texto de entrada que apresenta a seguinte estrutura:
    - Primeira linha: número de aeroportos da região.
    - Linhas seguintes: nome do aeroporto 1; nome do aeroporto 2; direção do voo; distância entre os aeroportos; duração do voo; horarios de partidas dos voos.
- Exemplo de arquivo de entrada:

```
3
CONFINS; GUARULHOS; 1; 606; 1:16; 7:00; 9:00; 18:00
CONFINS; GUARULHOS; -1; 606; 1:16; 8:00; 12:00; 17:00; 21:00
CONFINS; SANTOS DUMONT; 1; 480; 1:10; 7:30
GUARULHOS; SANTOS DUMONT; 1; 421; 0:30; 8:00; 18:00
```

## Classes Bases

Para atender às diversas exigências do projeto, foram criadas um total de 19 arquivos classes da linguagem Java, sendo 7 delas classes de teste, 1 classe simples de execução e 1 classe de exceção.

Para entendimento geral da forma como todo o projeto foi estruturado, a seguir apresentamos uma explicação simples para as classes criada mais importantes. Especificações detalhadas de cada classe e seus membros podem ser conferidas na documentação em código:

1. Vertice: vértice de um grafo. No nosso projeto, apenas os vértices têm responsabilidade sobre seus vizinhos e suas arestas adjacentes. Cada vértice apresenta um nome identificador *String id* e um dicionário (ou mapa) *Map<Vertice, PesoDaAresta> adjacentes* que guarda a relação dos vértices adjacentes com o peso para acessá-los. Não existe nenhuma classe para representar arestas e os grafos dirigidos e não dirigidos mantêm o mesmo tipo de vértice.

2. PesoDaAresta: peso da conexão entre vértices vizinhos. Seus atributos são *int distancia* (em quilômetros), *DuracaoDeVoo duracao* e *List<Horario> horarios*. Para diferenciar pesos de voos e pesos de rotas, criamos geradores com nomes diferentes e mantivemos os construtores da classe privados. O gerador *criaPesoDeRota* não recebe horários e atribui o valor *null* para o membro que os representa. Ao ser acessado, o atributo de horários checa se seu valor é nulo e lança uma exceção se for o caso.

3. Grafo: grafo não dirigido. Sua única responsabilidade é sobre os vértices que o compõem e a conexão destes com seus nomes identificadores. Ou seja, todos os seus métodos recebem apenas identificadores ao se referirem a um vértice. Seus atributos são *int numeroDeVertices* e *Map<String, Vertice> verticeMap*.

4. GrafoDirigido: grafo dirigido. Sua única diferença com relação à classe Grafo é uma re-implementação do método *void adicionaAresta*. Afinal, na nossa implementação, apenas o vértice de origem nos grafos dirigidos sabem da existência de suas arestas adjacentes.

5. MalhaAerea: representação de uma malha aérea. Ela apresenta um grafo dirigido de voos e um grafo não dirigido de rotas. Os dois devem, obviamente, apresentar os mesmo vértices. É nessa classe que se concentram os métodos de resolução dos problemas específicos passados.

## Modelagem das Soluções para Problemas Específicos

### Leitura do Arquivo de Entrada:

Para a aquisição dos dados da malha aérea vindos de um arquivo texto, nós criamos uma classe utilitária estática intitulada *GeradorDeMalhaAerea*. Seu único membro estático importante é o método *MalhaAerea geraMalhaAerea(String nomeDoArquivo)*. Esse método recebe o nome do arquivo texto e o procura na pasta *dados* do projeto usando a classe *Path* da própria biblioteca padrão do Java. Se nenhum arquivo é encontrado, uma malha vazia é gerada. Se um arquivo for encontrado, todas as suas linhas são lidas e armazenadas em um grafo de voo e outro de rota em um objeto *MalhaAerea*.

### Viagens de Menor Custo:

Para resolver esse problema, nós usamos do algoritmo de Dijkstra para determinar o menor caminho entre dois vértices e sua respectiva distância total. Desta forma, foi possível detectar qual rota entre aeroportos seriam usadas.

Seus devidos retornos em texto estão presentes na classe *Grafo*. O grande problema envolto neste algoritmo é a sua complexidade de modularização devido ao fato da natureza dos custos. Foram requisitados quatro custos diferentes possíveis: distância, tempo de voo, tempo de viagem e número de conexões. Cada um desses, além de armazenados de forma diferentes pela *MalhaArea*, precisam ser parametrizados por funções diferentes. Ou seja, mesmo que seja o caso em que se projetasse uma interface para se abstrair tais métodos, ainda assim teríamos de passar um parâmetro específico, seja pela assinatura ou pelo próprio método.

Foi feita a escolha de expressar individualmente cada custo em métodos diferentes, mesmo que estes fossem semelhantes, devido ao fato de como cada distância deve ser mapeada. O algoritmo foi implementado na classe *AlgoritmoDeDijkstra*.

### Determinação de Alcance de Aeroportos:

Nosso grupo chegou na conclusão que esse problema é de, basicamente, determinar os componentes conexos do grafo de rotas e seus vértices de corte. Primeiro deve-se determinar se ele é conexo (todo aeroporto consegue alcançar qualquer outro). Se ele for, é necessário pegar seus vértices de corte. Se ele não for, é necessário pegar os componentes conexos do grafo.

Como o problema requisitou mais uma informação vinda das rotas, nós criamos o método *pegaMensagemRelacionadaAConectividade* na classe *MalhaAerea*. Esse método retorna uma String com as informações de conectividade

do grafo. Para isso, ele cria uma instância da classe *AnalisadorDeConectividade* a partir do objeto de rotas. O analisador que tem, por sua vez, responsabilidade sobre as informações de conectividade do grafo. Ele funciona da seguinte forma:

1. Separa os vértices do grafo em seus respectivos componentes. Isso é feito a partir de uma travessia em profundidade recursiva por todos os vértices. A cada vértice passado, um identificar de componente é salvo em uma estrutura de dicionário (Map do Java). Os vértices de mesmo componentes apresentam identificadores de componente iguais.
2. Checa a conectividade do vértice. Se existir mais de um identificador de componente, o grafo não é conexo.
3. Se o grafo for conexo, é necessário pegar seus vértices de corte. Para isso, é criado um clone do grafo sem cada vértice, chegando se ele continua conexo. Importante ressaltar que, devido a natureza referencial do Java, não era possível simplesmente clonar o grafo (afinal, todas as estruturas internas dele estão conectadas, não há por onde começar a clonagem). Por isso, uma estrutura de matriz de adjacência foi criada para cada novo grafo testado.
4. Se o grafo não for conexo, pegar as informações de quais vértices estão em quais componentes conexos.

**<u>Determinação do Último Voo para Chegar no Destino em Horário Limite:</u>**

Apesar de que esse serviço do sistema deve ser acessado pelo método *pegaHorarioDoUltimoVooSemChegarAtrasado* da classe *MalhaAerea*, uma classe própria *AnalisadorDeHorarios* foi criada apenas para resolvê-lo.

Nossa solução consistiu em dois passos:

1. Pegar todas as opções de rotas entre origem e destino.

Isso é feito a partir do método *pegaCaminhoEntreVertices* da classe Grafo. O método recebe o identificador do primeiro e do último vértice do caminho, passando recursivamente pelos vizinhos de cada e botando em pilhas de caminho.

2. Escolher horário inicial mais tarde de saída dentre as rotas que possibilite a chegada antes do horário limite.

É checado, em cada caminho adquirido, o horário mais tarde de saída do primeiro aeroporto que possibilite uma chegada antes do horário limite no último aeroporto do caminho. Conforme os horários vão sendo determinados, eles vão sendo comparados para se descobrir o mais tarde de todos.

A análise de horários de saída é feita da seguinte forma:

- O método *pegaHorarioMaisTardeDeSaida* chama o método estático *analisaHorararioMaisTardeQueChegaNoDestino* da classe *AnalisadorDeHorarios*.

- A classe pega cada uma das opções de horário inicial e faz todo o caminho a partir dos aeroportos, sempre checando se o horário limite foi passado, se existem voos possíveis e se o dia não virou.
- Por fim, os horário considerados possíveis são comparados usando o método estático da classe *Horario pegaHorarioMaisTarde*.

**Determinação de Número de Aeronaves e Rotas Usadas por Uma Empresa de Carga:**

Para resolver esse problema, nós usamos do algoritmo de Prim para determinar a árvore mínima do grafo não dirigido de rotas. Dessa forma, foi possível detectar quais rotas entre aeroportos seriam usadas. O número de aviões nada mais é que o número de arestas (rotas) usadas.

O retorno do método *pegaEmpresaDeCaminhoMinimo* da classe *MalhaAerea* retorna um objeto da classe *EmpresaAereaDeCarga*, uma classe criada especificamente para manter os dados das rotas utilizadas e do número de aeronaves.

Para utilizarmos do algoritmo de prim, foi criada uma classe *AlgoritmoDePrim*, representando toda a estrutura de dados necessária para que o algoritmo funcione. Nossa maior dificuldade nesse caso foi que o algoritmo adiciona de um em um o vértice de menor custo, junto da sua aresta de ligação. Toda a estrutura do nosso projeto se baseia em vértice que mantém responsabilidade pelo seus próprios vizinhos. Ou seja, nós não criamos uma estrutura para se guardar arestas. Foi necessário sempre guardar o vértice que liga ao que será adicionado, para pegar o peso da ligação a partir dele.

## Testes Unitários

Para testar nossas classes e métodos, nós criamos testes unitários com a biblioteca JUnit 4. Não foram todos os métodos nem classes que foram testadas, apenas os que julgamos necessários.

## Exceções

Foi criada apenas uma classe exceção em todo o projeto. Ela é intitulada *RotaNaoEVooExcecao* e é lançada apenas quando um peso de uma ligação entre vértices de uma rota tenta acessar seus horários de vôo. Nossa arquitetura, nesse caso, foi de criar apenas uma classe genérica de *PesoDeAresta*. Ao ser inicializado, o peso pode receber um vetor de horários se tornando um peso de voo ou não, se tornando um peso de rota.