**PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS - PUC MINAS**
**BACHARELADO EM SISTEMAS DE INFORMAÇÃO**

# Trabalho Prático - Algoritmo em Grafos

Relatório Técnico

**Henrique Costa Marques**
**Lêda Lacerda Bombinho do Val**
**Yan Silva Braga**

Belo Horizonte - MG

Dezembro, 2025

## Parte 1: Logística

### 1. Introdução

O relatório descreve o desenvolvimento do Sistema de Otimização de Rotas Logísticas (SORL), solicitado pela diretoria da Entrega Máxima Logística S.A.. O objetivo do sistema é analisar a eficiência da malha logística da empresa, solucionando gargalos operacionais, custos elevados e dificuldades no planejamento de manutenção.

O projeto foi implementado na linguagem C# , processando dados de entrada no formato padrão DIMACS.

### 2. Modelagem Computacional da Rede

A rede logística foi modelada computacionalmente como um Grafo Direcionado e Ponderado, onde:

- **Vértices:** Representam os Centros de Distribuição (Hubs) ou Pontos de Entrega.
- **Arestas:** Representam as Rotas Rodoviárias entre os hubs.
- **Pesos:** Indicam o custo financeiro (R$) do transporte por aquela rota.
- **Capacidades:** Indicam o limite máximo diário (toneladas) da rota.

### 2.1. Estruturas de Dados e Análise de Densidade

Para atender ao requisito de eficiência e adaptação à topologia da rede, o sistema implementa a classe `Representacao`. Antes de instanciar o grafo, o algoritmo analisa a densidade da rede  seguindo o cálculo: Arestas / Vértices * (Vértices - 1). Se o resultado for maior que 0.25 o grafo escolhido é Matriz de adjacência, senão, lista de adjacência.

Foram adotadas as seguintes estratégias de armazenamento:

1. **Matriz de Adjacência (`GrafoMatriz`):** Utilizada para grafos densos, garantindo acesso imediato O(1) às arestas para verificação de conectividade.

2. **Lista de Adjacência (`GrafoLista`):** Utilizada para grafos esparsos, otimizando o consumo de memória.
    - *Observação Técnica:* A implementação da Lista utilizou a estrutura de **Dicionários** (`Dictionary<int, List<Aresta>>`) para garantir robustez contra identificadores de vértices não sequenciais ou esparsos.

### 3. Algoritmos e Soluções Implementadas

#### I. Roteamento de Menor Custo

Dado um grafo direcionado em que os vértices representam hubs logísticos e as arestas representam rotas possíveis, com peso associado ao custo financeiro de utilização, deseja-se determinar o trajeto de menor custo entre dois centros. Para isso nós utilizamos o algoritmo de Dijkstra.

Os pesos das arestas representam custos não negativos (tarifas, pedágios, consumo de combustível, etc.). Dessa forma, optamos pela escolha de Dijkstra para cálculo de caminhos mínimos, pois garante a obtenção do menor custo entre origem e destino com complexidade aceitável mesmo em redes de grande porte.

**Implementação.**

- A estrutura do grafo é acessada via interface IGrafo, permitindo utilizar tanto lista de adjacência quanto matriz de adjacência, dependendo da densidade do arquivo DIMACS.
- As distâncias (dist) e predecessores (pred) são armazenados em dicionários, o que permite trabalhar com identificadores de vértices 1-based sem depender de contiguidade rígida.
- É utilizada uma PriorityQueue<int,int> da biblioteca padrão para selecionar, a cada iteração, o vértice com menor custo acumulado, reduzindo o custo do passo de extração do mínimo.
- O algoritmo realiza o relaxamento das arestas saindo do vértice atual e atualiza dist[w] e pred[w] sempre que encontra um caminho mais barato.
- Ao final, caso não exista caminho entre origem e destino, o sistema registra no log a mensagem *“Não existe caminho entre X e Y”*; caso contrário,

reconstrói o caminho percorrendo o vetor de predecessores de trás para frente e retorna o custo total e a sequência de hubs, por exemplo:
Custo Total: 25 | Caminho: 1 -> 3 -> 6 -> 10.

**II. Capacidade Máxima de Escoamento**

Considerando as rotas entre hubs com capacidades máximas diárias (em toneladas, número de caminhões, etc.), nosso objetivo é calcular qual é o fluxo máximo que pode ser escoado entre um hub de origem (fonte S) e um hub de destino (sorvedouro T). Para isso escolhemos o algoritmo de Edmonds-Karp.

O algoritmo de Edmonds–Karp é uma especialização do método de Ford–Fulkerson que utiliza busca em largura (BFS) na rede residual para encontrar caminhos aumentantes de menor comprimento (em número de arestas). Dessa forma, nos garantimos:

- Terminação mesmo com capacidades arbitrárias;
- Implementação relativamente simples sobre uma matriz residual;
- Comportamento previsível em instâncias de porte médio, como os grafos DIMACS utilizados.

**Implementação**.

- Inicialmente, foi construída uma rede residual residual[u,v] a partir das capacidades das arestas (e.Capacidade) fornecidas pelo grafo.
- Em seguida, é executada repetidamente uma BFS (BfsCaminhoAumentante) na rede residual para encontrar um caminho de S até T com capacidade residual positiva em todas as arestas.
- Para cada caminho aumentante encontrado:
    - é calculado o gargalo delta (menor capacidade residual ao longo do caminho);
    - as capacidades residuais das arestas diretas são diminuídas e das arestas reversas são aumentadas;
    - o valor de delta é somado ao fluxoMaximo.

- O número de iterações do laço principal corresponde ao número de caminhos aumentantes utilizados.
- Caso nenhuma BFS consiga encontrar um caminho de S até T, o sistema registra e exibe a mensagem:
  *"Não existe caminho da origem S até o destino T na rede residual (Fluxo máximo = 0)."*
  Isso ajuda o usuário a identificar situações em que T está desconectado ou isolado na rede.

O resultado exibido inclui origem, destino, fluxo máximo calculado e quantidade de caminhos aumentantes:
Exemplo:

Origem (fonte S): 1 | Destino (sorvedouro T): 6 | Fluxo Máximo: 33 | Caminhos aumentantes usados: 4.

**III. Expansão da Rede de Comunicação (Fibra Óptica)**

Planejar a expansão da infraestrutura (por exemplo, instalação de fibra óptica ou ampliação de rotas fixas) conectando todos os hubs da rede com o menor custo total de instalação, sem criar ciclos desnecessários. Para isso nos utilizamos o algorimto de Prim (Árvore Geradora Mínima – MST).

O problema foi modelado como Árvore Geradora Mínima em um grafo ponderado e conectado. Entre os algoritmos clássicos (Prim e Kruskal), Prim foi escolhido por se adaptar bem à representação por listas de adjacência e permitir construção incremental da árvore a partir de um vértice inicial.

**Implementação**:

- O algoritmo inicia do vértice 1, definindo chave[1] = 0 e as demais chaves como infinito.
- Em cada iteração, escolhe-se o vértice ainda não visitado com menor valor de chave, que passa a integrar a árvore vigente.

- As arestas que saem desse vértice são analisadas, e, sempre que uma aresta (u,v) apresenta peso menor que a chave atual de v, os vetores chave[v] e pai[v] são atualizados.
- Ao final, pai[v] descreve as arestas da MST e a soma das chaves fornece o peso total da árvore.
- O sistema gera um log detalhado com as arestas selecionadas:
  Hub 3 → Hub 7 | Custo: 5, etc., além de registrar o peso total e observações sobre conectividade.

**IV. Agendamento de Manutenções sem Conflito**

Definir um cronograma de manutenções em hubs/rotas de modo que rotas adjacentes não sejam interditadas simultaneamente, minimizando o número de turnos utilizados. Para isso, nós utilizamos Coloração Gulosa de Grafos (Greedy Coloring).

Cada vértice representa um hub (ou segmento de rede) e cada cor representa um turno de manutenção. Como determinar o número cromático mínimo é um problema NP-Difícil, optou-se por uma heurística gulosa, que fornece uma solução viável em tempo polinomial.

**Implementação**:

- Foi construído um mapa de conflitos bidirecional: se existe uma aresta entre u e v, considera-se que esses vértices não podem ter a mesma cor (turno).
- Para cada vértice, são marcadas as cores já utilizadas pelos vizinhos e, em seguida, atribui-se a menor cor disponível.
- O sistema contabiliza o número total de cores utilizadas (turnos necessários) e gera um log com um cronograma sugerido, por exemplo:
  Hub 1: Turno 1, Hub 2: Turno 2

Essa abordagem não garante o número mínimo teórico de turnos, mas produz rapidamente um plano consistente e operacionalmente aplicável.

**V. Rota Única de Inspeção**

Avaliar a viabilidade de planejar uma rota única de inspeção que:

- **Cenário A (Euleriano):** percorra cada **rota** exatamente uma vez, retornando ao hub inicial;
- **Cenário B (Hamiltoniano):** visite todos os **hubs** exatamente uma vez, retornando, se possível, ao ponto de partida.

**Soluções adotadas:**

**Cenário A – Ciclo Euleriano em um grafo direcionado.**

- Verifica-se, inicialmente, se o grafo direcionado atende às condições necessárias e suficientes para a existência de um ciclo euleriano:
    - grau de entrada = grau de saída para todos os vértices;
    - grafo fortemente conectado.
- A conectividade forte é checada via **algoritmo de Kosaraju**, realizando uma DFS no grafo original e outra no grafo reverso.
- Quando as condições são satisfeitas, utiliza-se o **Algoritmo de Hierholzer** para construir o ciclo euleriano, consumindo as arestas ao longo do percurso e registrando o custo total.

**Cenário B – Rota Hamiltoniana Heurística.**

- Por se tratar de um problema NP-Completo, foi implementada uma heurística do **vizinho mais próximo**:
    - inicia-se em um hub escolhido pelo usuário;
    - em cada passo, seleciona-se a primeira rota disponível que leva a um hub ainda não visitado;
    - o processo continua até não haver mais vizinhos não visitados.
- O sistema indica se a rota obtida é **completa** (todos os hubs visitados) ou **parcial**, bem como o custo acumulado do percurso.

Ambos os cenários são documentados em log, com a sequência de hubs/rotas e seus respectivos custos, auxiliando o setor de inspeção a visualizar rotas viáveis na malha logística.

## 4. Análise dos Resultados

Foram utilizados sete grafos no formato DIMACS, representando cenários de rede com diferentes tamanhos e densidades, variando desde instâncias pequenas (como grafo01, com poucos vértices e arestas) até redes maiores e mais complexas (como grafo07, com dezenas de hubs e centenas de conexões).

Para cada arquivo DIMACS carregado, o sistema:

- Construiu dinamicamente a estrutura do grafo através da interface IGrafo, escolhendo lista de adjacência ou matriz de adjacência com base na densidade calculada (m / n(n−1)), o que garantiu boas performances tanto para grafos esparsos quanto densos;
- Executou os cinco módulos principais (roteamento, fluxo máximo, árvore geradora mínima, coloração e rota de inspeção), registrando saídas em arquivos de log log_grafo0X.txt:
    - caminhos mínimos entre pares de hubs e seus custos;
    - valores de fluxo máximo e número de caminhos aumentantes utilizados, incluindo os casos em que não existe caminho entre S e T;
    - conjunto de arestas da árvore geradora mínima e peso total;
    - número de turnos de manutenção necessários e atribuição de turno para cada hub;
    - existência (ou não) de ciclo euleriano e rota heurística hamiltoniana obtida.

Os testes mostraram que o SORL consegue processar corretamente tanto instâncias pequenas quanto grafos de maior escala, mantendo tempos de resposta adequados para uso interativo em laboratório e em cenários de planejamento.

## 5. Conclusão

O desenvolvimento do Sistema de Otimização da Rede Logística (SORL) atingiu integralmente os objetivos propostos no trabalho, uma vez que conseguimos atender todos os requisitos esperados.

A modelagem dos problemas de roteamento, fluxo de carga, expansão de rede, agendamento de manutenção e inspeção em termos de grafos permitiu aplicar

algoritmos clássicos de Teoria dos Grafos de forma direta e coerente com o contexto empresarial.

A utilização da interface IGrafo e da classe Representacao para selecionar dinamicamente a estrutura de dados (matriz ou lista) tornou a solução mais flexível e escalável, adaptando-se a diferentes perfis de instâncias.

A escolha de algoritmos consagrados (Dijkstra, Edmonds–Karp, Prim, coloração gulosa, Hierholzer, Kosaraju e vizinho mais próximo) proporcionou um bom equilíbrio entre qualidade das soluções e complexidade computacional.

A geração de logs detalhados para cada módulo facilita tanto a análise dos resultados quanto a auditoria das decisões sugeridas pelo sistema, aproximando o projeto de uma ferramenta real de apoio ao planejamento logístico.

Assim, o SORL se mostra uma plataforma consistente para estudo e simulação de problemas de otimização em redes logísticas, servindo tanto como instrumento didático quanto como base para futuras extensões, como inclusão de restrições de capacidade em múltiplos períodos, janelas de tempo ou integração com dados reais da empresa.

## Parte 2: Desafio Beecrowd

### Introdução

Essa etapa no trabalho prático constituiu em realizar o problema “A casa das 7 mulheres - 3350” disponibilizado na plataforma Beecrowd.

Link da solução na plataforma: https://judge.beecrowd.com/en/runs/code/47543901

### Problema

Temos 2N províncias numeradas de 1 a 2N, agrupadas em N pares de províncias-gêmeas: (1,2),(3,4),…,(2N−1,2N). Para cada província I, a entrada informa exatamente 6 outras províncias diretamente alcançáveis a partir de I, por meio de estradas dirigidas.

O objetivo é escolher um conjunto de províncias S para receber diretórios, respeitando duas regras:

1. **Restrição dos pares de gêmeas:** para cada par de províncias-irmãs, no máximo uma pode receber diretório.
2. **Restrição de cobertura por vizinhos:** para cada província I que não recebe diretório, deve haver pelo menos duas províncias em S diretamente alcançáveis a partir de I (entre os seus 6 vizinhos de saída).

O enunciado ainda garante que existe uma solução “forte”, chamamos de T, com propriedades mais rígidas (cada província tem cabeça satisfeita e pelo menos 3 vizinhos, ou 4 vizinhos na cauda), o que sugere que o conjunto de soluções válidas para a nossa condição mais fraca é grande.

O desafio está em encontrar uma dessas soluções em tempo polinomial, respeitando o limite de tempo de 1 segundo e o número máximo de províncias.

### 3. Solução

### 3.1. Representação do grafo e dos pares

- O grafo é representado por uma lista de adjacência:

    - grafoSaida[i] armazena os 6 vizinhos alcançáveis a partir do vértice iii;
- Os pares de vértices-gêmeos são dados implicitamente:
    - vértices 1 e 2 formam um par, 3 e 4 outro, e assim por diante;
    - a função Gemea(i) devolve o vértice gêmeo de I.

Mantemos um vetor booleano diretorioEscolhido[i] que indica se o vértice I foi escolhido para receber diretório. Por construção, o algoritmo sempre tenta manter que, em cada par, apenas um vértice está marcado como verdadeiro.

### 3.2. Restrição local por vértice (pontuação)

Para cada vértice I, há uma **restrição local**:

- se I não recebe diretório (diretorioEscolhido[i] == false), então pelo menos **2** dos seus 6 vizinhos de saída devem receber diretório.

No código, essa condição é monitorada por meio de uma pontuação para cada vértice, pontuacaoClausula[i], definida assim:

- começamos com cont = 0;
- para cada vizinho v de I, se diretorioEscolhido[v] == true, incrementamos cont;
- se o próprio vértice I possui diretório (diretorioEscolhido[i] == true), somamos mais 2 ao contador.

Interpretação:

- se o próprio vértice está em S, a pontuação é pelo menos 2 → a restrição local está automaticamente satisfeita;
- se o vértice não está em S, a pontuação é exatamente o número de vizinhos em S. A condição do problema exige que essa pontuação seja pelo menos 2.

Chamamos de vértice problemático (ou cláusula ruim) todo vértice *I* cuja pontuação é:

pontuacaoClausula[i] <= 1

Ou seja: vértice não direcionado e com zero ou apenas um vizinho com direcionado.

Mantemos:

- um vetor clausulasRuins que guarda os índices dos vértices problemáticos;
- um vetor posicaoNaListaRuins[i] para saber, em O(1), em que posição ele está na lista;
- um contador quantidadeRuins.

Isso nos permite sortear rapidamente um vértice problemático a cada iteração.

### 3.3. Estrutura auxiliar de incidência

Quando trocamos a escolha de um vértice de um par (por exemplo, tiramos o diretório de *u* e colocamos em seu gêmeo *v*), isso pode afetar:

- a restrição local do próprio *u* e de *v*(se eles ganham ou perdem direção no grafo);
- as restrições locais de todos os vértices que têm aresta de saída para *u* ou para *v*(porque um vizinho deles entrou ou saiu de *S*).

Para atualizar essas pontuações de forma eficiente, o programa mantém, para cada vértice *w*:

clausulasDaCaudaPorVertice[w] = lista de vértices *l* tais que existe aresta *l* -> *w*;

Ou seja, clausulasDaCaudaPorVertice[w] contém todos os vértices cuja restrição local depende de *w* como vizinho de saída. Assim, quando um vértice passa a ser direcionado ou deixa de ter, atualizamos apenas as restrições diretamente impactadas.

### 3.4. Busca local aleatória (passeio em um grafo de estados)

Em vez de explorar todas as escolhas possíveis de vértices (o que seria exponencial), a solução usa uma no espaço de configurações. Cada configuração é um conjunto S que:

- contém exatamente um vértice de cada par de gêmeos.

Podemos imaginar um grafo de estados, em que:

- cada vértice do grafo de estados é uma escolha possível de *S*;
- há uma aresta entre duas configurações se elas diferem apenas na escolha de um par de gêmeos

O algoritmo implementa um passeio aleatório nesse grafo de estados:

1. **Inicialização**
    - para cada par de gêmeos, escolhemos aleatoriamente um dos dois vértices para entrar em *S*;
    - calculamos a pontuação de todas as restrições locais (pontuacaoClausula[i]) e montamos a lista inicial de vértices problemáticos (clausulasRuins).

2. **Enquanto houver vértices problemáticos**:
    a. Escolhemos aleatoriamente um vértice problemático *l* da lista.
    b. Aplicamos uma das duas operações:
    - **Operação 1 (probabilidade 1/6)**
        i. Inverter a escolha no par do próprio vértice *l*:
        ii. se o gêmeo de *l* está em *S*, retiramos o gêmeo e colocamos *l* em *S*.
    - **Operação 2 (probabilidade 5/6)**
        i. Olhamos os 6 vizinhos de saída de *l* e coletamos aqueles que não estão em *S*.
        ii. Escolhemos um desses vizinhos aleatoriamente e invertemos a escolha no par dele (retirando o gêmeo e colocando esse vizinho em *S*).

3. **A cada inversão, atualizamos:**
    - o vetor diretorioEscolhido[] para o par envolvido;
    - as pontuações das restrições locais dos vértices afetados;
    - a lista de vértices problemáticos (clausulasRuins), incluindo ou removendo vértices conforme sua pontuação passa a ser >1 ou ≤1.

Esse processo é um passeio aleatório em um grafo de estados de dimensão N. Além disso, devido à garantia extra de que existe uma solução “forte”, o algoritmo converge rapidamente para uma configuração em que não há mais vértices problemáticos.

### 3.5. Verificação final

Quando a lista de vértices problemáticos (clausulasRuins) fica vazia, significa que:

- para todo vértice *I* sem caminho, ele atinge pelo menos 2 vizinhos com diretório.

O programa ainda faz uma checagem final para:

- garantir que em cada par de gêmeos existe no máximo um vértice em *S*;
- verificar novamente a condição de vizinhos para cada vértice não escolhido.

Se tudo estiver correto, a solução *S* (conjunto de vértices com diretorioEscolhido[i] == true) é impressa.