Algoritmos e Estruturas de Dados II

Capítulo da Aula 26: Caminhos Eulerianos e Hamiltonianos

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br
CEFET-MG - Campus Timóteo

Junho de 2026

Parte 1: Caminhos Eulerianos

1. O Problema Histórico das Pontes de Königsberg (1736)

A cidade de Königsberg (hoje Kaliningrado, Rússia) era cortada pelo rio Pregel e possuía **sete pontes** que conectavam quatro regiões distintas: duas margens, uma ilha central e uma península.

O desafio popular da época era: *é possível fazer um passeio pela cidade cruzando cada ponte exatamente uma vez?* O matemático **Leonhard Euler** (1736) foi o primeiro a **provar matematicamente** que é impossível — e ao fazer isso, criou as bases da **Teoria dos Grafos**, tornando-se o problema inaugural da área.

A ideia de Euler foi modelar o problema como um grafo: cada região vira um vértice e cada ponte vira uma aresta. A pergunta passa a ser: existe um caminho que percorra cada **aresta** exatamente uma vez?

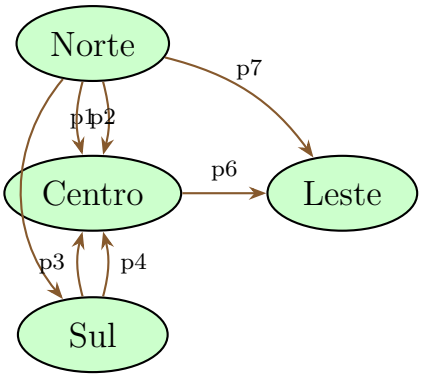


Figure 1: As 7 pontes de Königsberg modeladas como grafo. Cada região é um vértice; cada ponte, uma aresta. Norte e Sul têm grau 3; Centro tem grau 5; Leste tem grau 3. Todos os graus são ímpares — não existe circuito Euleriano!

• Teoria

Euler provou que este tipo de percurso é impossível **sem** precisar testar todas as combinações. Sua prova foi baseada em uma propriedade estrutural simples dos vértices, fundando a Teoria dos Grafos em 1736.

2. Definições: Caminho e Ciclo Euleriano

- **Caminho Euleriano:** Caminho que percorre **cada aresta exatamente uma vez**. Não precisa retornar ao vértice de origem.
- **Ciclo (Circuito) Euleriano:** Caminho Euleriano que **começa e termina no mesmo vértice**.

Atente para a diferença com os conceitos Hamiltonianos: o Euleriano trata de **arestas**, e o Hamiltoniano trata de **vértices**.

3. Teorema de Euler: Condição de Existência

• Teorema de Euler (grafos não-direcionados)

Ciclo Euleriano existe $\iff$ o grafo é **conexo E** todos os vértices têm **grau par**.

Caminho Euleriano existe $\iff$ o grafo é conexo **E** tem **exatamente 0 ou 2 vértices de grau ímpar**. (Com 2 vértices ímpares, o caminho começa em um e termina no outro.)

A verificação desta condição é trivialmente $O(V + E)$: uma busca em largura ou profundidade para conectividade, mais uma contagem de graus.

Por que a condição de grau par é necessária?

Pense em qualquer vértice v interno ao ciclo Euleriano: cada vez que chegamos a v por uma aresta, precisamos sair por outra. Portanto, as arestas de v aparecem em pares (**entrada + saída**), exigindo grau par. Para o vértice de início/fim do ciclo, o mesmo raciocínio se aplica (começa por uma aresta e termina voltando por outra). Portanto, **todos** os vértices precisam de grau par.

Grafos Direcionados

Em **dígrafos** (grafos direcionados), as condições mudam:

- **Circuito Euleriano (dígrafo):** Existe $\Leftrightarrow$ o grafo é **fortemente conexo** E para cada vértice: grau-entrada = grau-saída.
- **Caminho Euleriano (dígrafo):** Existe $\Leftrightarrow$ há exatamente um vértice com saída − entrada = 1 (origem) e um com entrada − saída = 1 (destino), e os demais são equilibrados.

▷ **Aplicação: Sequências de De Bruijn**

Uma sequência de De Bruijn de ordem k sobre um alfabeto de tamanho n é uma sequência circular que contém cada possível substring de tamanho k exatamente uma vez. Este problema se reduz a encontrar um **Circuito Euleriano** num dígrafo onde cada nó é uma substring de tamanho $k-1$ e cada aresta representa concatenar um símbolo.

4. Algoritmo de Hierholzer $O(E)$

Uma vez que sabemos que o Ciclo Euleriano **existe**, como encontrá-lo eficientemente? O Algoritmo de Hierholzer (1873) faz isso em $O(E)$.

Ideia Central

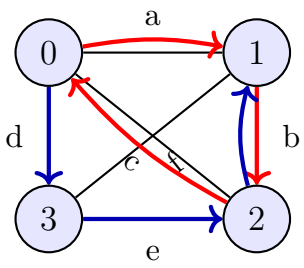
1. Começe de qualquer vértice v com arestas disponíveis.
2. Siga arestas arbitrariamente, **removendo-as** do grafo conforme são usadas, até voltar a v (formando um sub-ciclo).
3. Se ainda há arestas não visitadas, encontre qualquer vértice do sub-ciclo atual que ainda possua arestas disponíveis. Repita o processo a partir desse vértice para formar um novo sub-ciclo.
4. **Funda** os sub-ciclos ao ciclo principal.
5. Repita até que todas as arestas sejam incluídas.

Implementação Java do Hierholzer

```java
import java.util.*;

public class Hierholzer {
    // adj[u] = lista de {vizinho, id_aresta}
    static List<int[]>[] adj;
    static boolean[] usado; // aresta ja usada?
    static LinkedList<Integer> ciclo = new LinkedList<>();

    static void dfs(int u) {
        while (!adj[u].isEmpty()) {
            int[] aresta = adj[u].remove(adj[u].size() - 1);
            int v = aresta[0], id = aresta[1];
            if (usado[id]) continue;
            usado[id] = true;
            dfs(v);
        }
        ciclo.addFirst(u); // insere na frente ao retornar (pos
            -ordem)
```



Sub-ciclo 1 (vermelho): 0-1-2-0
Sub-ciclo 2 (azul): expansão a partir de 0

Figure 2: Hierholzer: sub-ciclos são formados e depois fundidos para obter o Ciclo Euleriano completo.

```
18      }
19
20      @SuppressWarnings("unchecked")
21      public static List<Integer> encontrarCiclo(int V, int[][]
           arestas) {
22          adj = new List[V];
23          for (int i = 0; i < V; i++) adj[i] = new ArrayList<>();
24          usado = new boolean[arestas.length];
25
26          for (int i = 0; i < arestas.length; i++) {
27              int u = arestas[i][0], v = arestas[i][1];
28              adj[u].add(new int[]{v, i});
29              adj[v].add(new int[]{u, i}); // grafo nao-
                   direcionado
30          }
31
32          dfs(0); // comeca no vertice 0
33          return new ArrayList<>(ciclo);
34      }
35  }
```

△ Importante

Importante: A implementação acima usa DFS iterativa (removendo arestas pelo final da lista). A inserção na **frente** da lista `ciclo` durante o retorno da DFS (pós-ordem) é o que garante a ordem correta do ciclo Euleriano. Não tente imprimir na ordem de visita direta — o ciclo resultante seria incorreto.

• Teoria

Complexidade de Hierholzer: $O(V + E)$. Cada aresta é visitada e removida exatamente uma vez. O uso da lista duplamente ligada para o ciclo garante inserção em $O(1)$.

5. O Problema do Carteiro Chinês

Quando o grafo **não é Euleriano** (tem vértices de grau ímpar), ainda podemos querer percorrer todas as arestas com o **menor custo total**. Este é o **Problema do Carteiro Chinês** (Chinese Postman Problem).

Algoritmo:

1. Encontre todos os vértices de **grau ímpar** (sempre em número par, pelo Lema do Aperto de Mãos).
2. Compute os **caminhos mínimos** entre todos os pares de vértices ímpares (Floyd-Warshall ou Dijkstra repetido).
3. Calcule o **emparelhamento perfeito de peso mínimo** entre eles (algoritmo de Blossom de Edmonds, $O(V^3)$).
4. **Duplique** as arestas dos caminhos mínimos correspondentes a cada par emparelhado. Agora todos os vértices têm grau par.
5. Aplique Hierholzer no grafo resultante.

▷ Aplicação: Rota de Varredura de Ruas

Um veículo de limpeza deve percorrer todas as ruas de um bairro sem deixar nenhuma de fora. Se o grafo de ruas tem vértices de grau ímpar (cruzamentos com número ímpar de ruas), algumas ruas precisarão ser repetidas. O Problema do Carteiro Chinês minimiza o total de percurso extra necessário.

Parte 2: Caminhos Hamiltonianos e TSP

6. O Jogo Icosiano de Hamilton (1857)

O matemático irlandês **William Rowan Hamilton** propôs em 1857 o "Icosian Game": dado um dodecaedro (sólido com 12 faces pentagonais e 20 vértices), encontrar um ciclo

que passe por **todos os vértices exatamente uma vez** e retorne ao início. Hamilton patenteou o jogo como um quebra-cabeça de madeira comercializado na Inglaterra.

- **Caminho Hamiltoniano:** Caminho que visita cada **vértice** exatamente uma vez (sem obrigatoriedade de retornar ao início).
- **Ciclo Hamiltoniano:** Caminho Hamiltoniano que **retorna ao vértice de origem**.

△ Importante

O Grande Abismo de Complexidade: Enquanto verificar e encontrar um Ciclo Euleriano é $O(E)$ (linear!), verificar se um Ciclo Hamiltoniano **existe** em um grafo geral é **NP-Completo**. Não existe algoritmo polinomial conhecido para este problema (a menos que P = NP).

7. Por que Hamiltoniano é Tão Mais Difícil?

A diferença fundamental está no que é contado:

Característica	Euleriano	Hamiltoniano
O que percorre	Cada **aresta** 1x	Cada **vértice** 1x
Condição de existência	Sim (grau par)	Não (caso geral)
Complexidade (encontrar)	$O(E)$ (Hierholzer)	$O(2^N \cdot N)$ (DP)
Classe do problema	P	NP-Completo
Critério suficiente	Grau par (todos)	Ore, Dirac ...

Para o Euleriano, a presença de uma aresta não visitada é uma informação **local** (observável no vértice). Para o Hamiltoniano, saber se um vértice foi visitado requer informação **global** sobre todo o caminho percorrido até aquele momento. É essa assimetria que torna o problema exponencialmente mais difícil.

• Teoria

A distinção entre Euleriano (P) e Hamiltoniano (NP-Completo) é um dos exemplos mais didáticos de como uma pequena mudança na definição de um problema pode alterar radicalmente sua classe de complexidade computacional.

8. Condições Suficientes para Ciclo Hamiltoniano

Embora não exista um critério geral "se e somente se", há condições **suficientes** (mas não necessárias) que garantem a existência:

Teorema de Dirac (1952)

Se G tem $n \geq 3$ vértices e todo vértice tem grau $\geq n/2$, então G possui Ciclo Hamiltoniano.

Teorema de Ore (1960)

Se para todo par de vértices **não-adjacentes** u, v: $\deg(u) + \deg(v) \geq n$, então existe Ciclo Hamiltoniano. (Note que Dirac é um caso especial de Ore, com $n/2 + n/2 = n$.)

△ Importante

Cuidado: São condições suficientes, **não necessárias**. Um ciclo simples C_n (com n vértices) é Hamiltoniano mas tem grau 2 para todos os vértices — não satisfaz Dirac para $n > 4$.

9. TSP — O Problema do Caixeiro Viajante

O TSP (Traveling Salesman Problem) é a versão de **otimização** do Hamiltoniano:

Dado um grafo completo ponderado, encontrar o Ciclo Hamiltoniano de ***menor peso total****.*

Como é NP-Hard, não existe algoritmo polinomial exato para instâncias grandes. As abordagens dependem de N:

- $N \leq 20$: **Programação Dinâmica com Bitmask** — $O(2^N \cdot N^2)$
- $N \leq 12$: Backtracking com poda
- N grande: Heurísticas (2-OPT, Ant Colony, Genético) ou Aproximações (Christofides: 1.5× ótimo para TSP métrico)

• Teoria

Para $N = 25$: número de ciclos distintos $\approx (25-1)!/2 \approx 3 \times 10^{23}$. Nem mesmo todos os computadores do mundo juntos conseguem explorar isso em tempo razoável. Heurísticas e aproximações são as únicas abordagens práticas.

10. DP com Bitmask: A Técnica Central

A ideia-chave é representar o **conjunto de cidades visitadas** como um inteiro (máscara de bits), onde o i-ésimo bit vale 1 se a cidade i foi visitada.

Definição do Estado:

$dp[\texttt{mask}][u]$ = menor custo para visitar exatamente as cidades em `mask`, terminando na cidade u

Transição: Para cada cidade v ainda não visitada em `mask`:

$$dp[\texttt{mask} \mid (1 \ll v)][v] = \min\Big(dp[\texttt{mask} \mid (1 \ll v)][v],\ dp[\texttt{mask}][u] + \text{dist}[u][v]\Big)$$

Resposta Final (fechando o ciclo de volta à cidade 0):

$$\text{ans} = \min_{u \neq 0}\Big(dp[\texttt{ALL}][u] + \text{dist}[u][0]\Big), \quad \texttt{ALL} = 2^N - 1$$

Operações com Bitmask

Operação	Java	Significado
Bit i está ativo?	`(mask & (1<<i)) != 0`	Cidade i visitada?
Ativar bit i	`mask | (1<<i)`	Visitar cidade i
Desativar bit i	`mask ^ (1<<i)`	Remover cidade i
Máscara completa	`(1<<N) - 1`	Todas visitadas
Contar bits ativos	`Integer.bitCount(mask)`	Quantas visitadas

▷ Trace para $N = 3$

Cidades 0, 1, 2. Distâncias: $d(0,1) = 10$, $d(0,2) = 15$, $d(1,2) = 20$.

- $dp[001][0] = 0$ (base: começa na cidade 0)
- $dp[011][1] = 10$ ($0 \to 1$, custo 10)
- $dp[101][2] = 15$ ($0 \to 2$, custo 15)
- $dp[111][1] = dp[101][2] + d(2,1) = 15 + 20 = 35$
- $dp[111][2] = dp[011][1] + d(1,2) = 10 + 20 = 30$
- ans $= \min(35 + 10,\ 30 + 15) = \min(45, 45) = \mathbf{45}$
- Ciclo ótimo: $0 \to 1 \to 2 \to 0$ ou $0 \to 2 \to 1 \to 0$ (ambos custam 45).

11. Implementação Java: TSP com DP Bitmask

```java
import java.util.*;

public class TSP {
    static final int INF = Integer.MAX_VALUE / 2;

    public static int solveTSP(int[][] dist) {
        int N = dist.length;
        int ALL = (1 << N) - 1;
        int[][] dp = new int[1 << N][N];

        for (int[] row : dp) Arrays.fill(row, INF);
        dp[1][0] = 0; // comeca na cidade 0, mask=0...01

        for (int mask = 1; mask <= ALL; mask++) {
            for (int u = 0; u < N; u++) {
                // u deve estar em mask
                if ((mask & (1 << u)) == 0) continue;
                if (dp[mask][u] == INF) continue;

                // Expande para cidades nao visitadas
```

```
                    for (int v = 0; v < N; v++) {
                        if ((mask & (1 << v)) != 0) continue; // v
                            ja visitado
                        int newMask = mask | (1 << v);
                        int novoCusto = dp[mask][u] + dist[u][v];
                        if (novoCusto < dp[newMask][v])
                            dp[newMask][v] = novoCusto;
                    }
                }
            }

            // Fecha o ciclo voltando a cidade 0
            int ans = INF;
            for (int u = 1; u < N; u++)
                if (dp[ALL][u] != INF && dp[ALL][u] + dist[u][0] <
                    ans)
                    ans = dp[ALL][u] + dist[u][0];
            return ans;
        }
    }
```

• Teoria

Complexidade:

- **Tempo:** $O(2^N \cdot N^2)$ — para $N = 20$: $\approx 4 \times 10^8$ operações (limite prático de 1-2s).
- **Espaço:** $O(2^N \cdot N)$ — para $N = 20$: ≈ 80 MB.
- Para $N = 25$: tempo $\approx 10^{10}$, espaço ≈ 3 GB. Impraticável!

12. Reconstruindo o Caminho Ótimo

Para recuperar a rota (não apenas o custo), é necessário manter um vetor de **pais** durante a DP:

```
// Adicionar ao solveTSP: tambem mantenha pai[mask][v] = u
int[][] pai = new int[1 << N][N];
for (int[] row : pai) Arrays.fill(row, -1);

// Dentro do loop de expansao, substitua a atualizacao por:
if (novoCusto < dp[newMask][v]) {
    dp[newMask][v] = novoCusto;
    pai[newMask][v] = u; // veio de u
}

// Reconstrucao (apos encontrar melhorUltimo = u que minimiza
    ans):
List<Integer> caminho = new ArrayList<>();
```

```java
int mask = ALL, cur = melhorUltimo;
while (cur != -1) {
    caminho.add(cur);
    int prev = pai[mask][cur];
    mask ^= (1 << cur);
    cur = prev;
}
Collections.reverse(caminho);
caminho.add(0); // fecha o ciclo
```

13. Heurísticas para TSP Grande

Quando N é grande demais para a DP exata, usamos heurísticas:

Vizinho Mais Próximo (Greedy)

Começa em uma cidade qualquer e sempre vai para a **cidade mais próxima ainda não visitada**. Rápido ($O(N^2)$), mas pode gerar soluções muito ruins (não há garantia de qualidade).

2-OPT (Melhoria Local)

Parte de qualquer ciclo Hamiltoniano e iterativamente tenta melhorá-lo: remove duas arestas (A, B) e (C, D), reconectando como (A, C) e (B, D). A troca é aceita se reduz o custo total. Repete até que nenhuma troca de dois pares melhore a solução. Complexidade: $O(N^2)$ por iteração. Converge para um ótimo **local**.

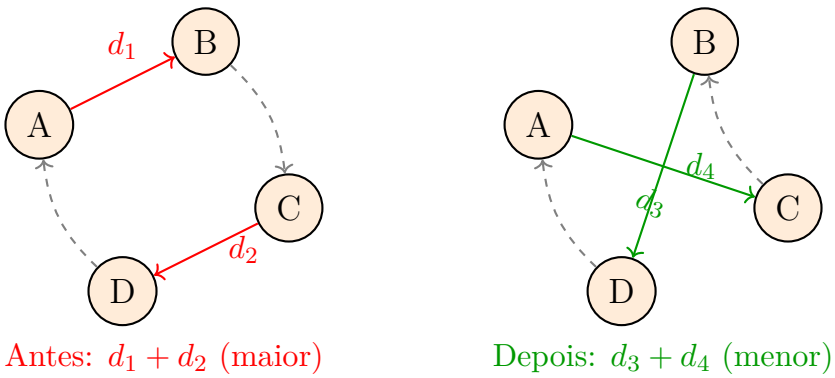


Figure 3: Operação 2-OPT: remove duas arestas e reconecta de forma diferente, reduzindo o custo total.

Algoritmo de Christofides (Aproximação)

Para o **TSP métrico** (com desigualdade triangular, $d(A,C) \leq d(A,B) + d(B,C)$), o Algoritmo de Christofides garante uma solução no máximo 1.5× o ótimo:

1. Calcule a **Árvore Geradora Mínima** (MST) do grafo.
2. Encontre os vértices de grau ímpar na MST (sempre número par).
3. Calcule o **emparelhamento perfeito de peso mínimo** entre esses vértices ímpares.
4. Una MST + emparelhamento para criar um multigrafo com todos os graus pares.
5. Encontre o **Circuito Euleriano** nesse multigrafo (Hierholzer).
6. "Atalhe" vértices repetidos (pela desigualdade triangular, o custo não aumenta).

Complexidade: $O(N^3)$ (dominado pelo emparelhamento). Garantia: solução $\leq 1.5\times$ ótimo.

Parte 3: Aplicações

14. Aplicações de Caminhos Eulerianos

- **Inspecção de Dutos e Redes Ferroviárias:** Um robô ou inspetor deve verificar cada trecho de uma rede de gasodutos ou trilhos exatamente uma vez. Modelagem direta como Ciclo Euleriano.
- **Sequenciamento de DNA (Montagem de Genomas):** Fragmentos de DNA são modelados como arestas de um grafo de De Bruijn. Montar o genoma = encontrar o Circuito Euleriano.
- **Varredura de Áreas (Robós de Limpeza):** Robôs Roomba e similares precisam cobrir toda área de um cômodo percorrendo cada setor. O planejamento de rota é equivalente ao Carteiro Chinês.
- **Trajetoria de Impressoras 3D e Plotters:** A cabeça de impressão deve percorrer todos os segmentos de uma figura sem repetir movimentos. Problema Euleriano em grafos direcionados.

15. Aplicações do TSP e Hamiltoniano

- **Logística e Entregas (Last Mile):** Empresas como Amazon e FedEx otimizam rotas de entrega minimizando a distância total percorrida. Variantes do TSP com janelas de tempo (TSPTW).
- **Manufatura de PCB (Placas de Circuito Impresso):** Brocas CNC devem perfurar centenas de furos numa placa. A ordem de visita dos furos (TSP) impacta diretamente o tempo de produção.

- **Escalonamento de Tarefas:** Ordenar tarefas de forma que o custo de transição entre elas seja minimizado. Modelado como TSP no espaço de tarefas.
- **Tourismo e Planejamento de Visitas:** Encontrar a ordem ótima para visitar pontos turísticos de uma cidade minimizando deslocamento. Caso clássico de TSP.

Parte 4: Exercícios

16. Exercícios Conceituais

1 **(Fundamentação)** Prove formalmente o Teorema de Euler: um grafo conexo possui Ciclo Euleriano se e somente se todos os vértices têm grau par.
Dica: Prove a necessidade (grau par ⇒ ciclo) e a suficiência (ciclo ⇒ grau par) separadamente. Para a suficiência, use o Algoritmo de Hierholzer como construção.

2 **(Complexidade)** Explique, usando argumentos de Teoria da Complexidade, por que TSP é NP-Hard enquanto Euleriano é P. Qual é a diferença estrutural que causa isso?

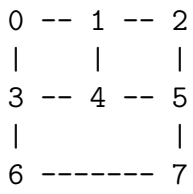
3 **(Teoremas de Condição)** Dê um exemplo de grafo que:

- Satisfaz o Teorema de Dirac e tem Ciclo Hamiltoniano.
- Não satisfaz Dirac, mas ainda tem Ciclo Hamiltoniano.
- Não satisfaz Dirac e não tem Ciclo Hamiltoniano.

4 **(Heurísticas)** Compare os algoritmos 2-OPT e Vizinho Mais Próximo para TSP. Em que cenários cada um é preferível? Dê um exemplo de grafo onde o Vizinho Mais Próximo gera uma solução significativamente pior que o ótimo.

17. Exercícios Analíticos

1 **(Euler)** Para o grafo abaixo, determine se existe Ciclo Euleriano, Caminho Euleriano ou nenhum. Se existir, encontre-o.



Resposta: Calcule os graus de todos os vértices. Vértices com grau ímpar?

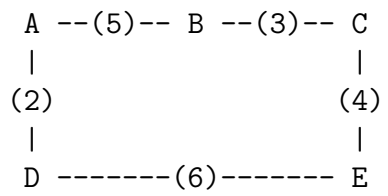
2 **(TSP Trace)** Aplique a DP com Bitmask ao grafo de 4 cidades com matriz de distâncias:

$$D = \begin{pmatrix} 0 & 10 & 15 & 20 \\ 10 & 0 & 35 & 25 \\ 15 & 35 & 0 & 30 \\ 20 & 25 & 30 & 0 \end{pmatrix}$$

Mostre a tabela $dp[\texttt{mask}][u]$ completa e o caminho ótimo com seu custo. *(Resposta esperada: custo 80, rota* $0 \to 1 \to 3 \to 2 \to 0$*.)*

3 **(Análise)** Para o TSP com N cidades, quantos **estados distintos** existem na DP com Bitmask? Qual é o limite prático de N considerando: (a) limite de tempo de 2 segundos ($\approx 4 \times 10^8$ operações); (b) limite de memória de 256 MB?

4 **(Carteiro Chinês)** Para o grafo:



Encontre o percurso de custo mínimo que visita todas as arestas.
Dica: Identifique os vértices de grau ímpar e calcule qual par duplicar.

18. Exercícios de Programação

1 **(Hierholzer)** Implemente o Algoritmo de Hierholzer em Java. A entrada será um grafo não-direcionado conexo. Antes de executar, verifique se o Ciclo Euleriano existe (graus pares). Se não existir, indique o motivo. Teste com:

- Grafo com todos graus pares (deve imprimir o ciclo).
- Grafo com exatamente 2 vértices ímpares (deve imprimir o caminho Euleriano).
- Grafo com 4+ vértices ímpares (deve indicar que é impossível).

2 **(TSP Completo)** Implemente o TSP com DP Bitmask incluindo:

- Cálculo do custo mínimo do ciclo.
- Reconstrução e impressão do caminho ótimo.
- Teste com $N = 4$ e a matriz do Exercício 17.2.
- Comparação com força bruta para $N \leq 10$.

3 **(2-OPT)** Implemente o algoritmo 2-OPT para melhoria de solução do TSP. Use o Vizinho Mais Próximo como solução inicial. Aplique 2-OPT até que nenhuma melhoria seja encontrada. Meça e compare a qualidade com o ótimo da DP para $N \leq 15$.

4 **(Desafio — Carteiro Chinês Simplificado)** Implemente o algoritmo do Carteiro Chinês para grafos com até 10 vértices ímpares. Use Dijkstra para os caminhos mínimos entre os vértices ímpares e force bruta para o emparelhamento (viável para até 10 vértices ímpares pois $10!! = 945$ emparelhamentos). Imprima o percurso completo (incluindo arestas duplicadas) e o custo total.