Implementação e Análise do Problema do Ciclo Hamiltoniano

Ítalo Jefferson Lima dos Santos
Instituto Federal de Educação, Ciência e Tecnologia da Paraíba
Campus Santa Rita
Curso de Análise e Desenvolvimento de Sistemas
italo.jefferson@academico.ifpb.edu.br
GitHub: https://github.com/Italo520/Projeto_Disciplina_EDA.git

Dezembro 2025

Resumo

Este relatório técnico apresenta a implementação e a análise experimental do problema do Ciclo Hamiltoniano, um problema clássico da classe NP-Completo [Garey 1979]. Primeiramente, é feita uma contextualização teórica do problema, discutindo sua formulação como problema de decisão e as implicações de pertencer à classe NP-Completo em termos de tempo de execução e escalabilidade. Em seguida, descreve-se a metodologia utilizada para o desenvolvimento de uma solução exata baseada em backtracking [Cormen 2012], incluindo as principais estruturas de dados empregadas e as decisões de projeto. Por fim, são apresentados e analisados resultados empíricos de tempo de execução para diferentes tamanhos de entrada, evidenciando de forma clara o crescimento exponencial do esforço computacional e confirmando rigorosamente a teoria da complexidade associada ao problema.

1 Introdução

O problema do Ciclo Hamiltoniano é um dos problemas clássicos de teoria dos grafos e de complexidade computacional [Cormen 2012]. De forma geral, ele consiste em determinar se, dado um grafo, existe um ciclo simples que visite cada vértice exatamente uma vez e retorne ao vértice inicial. Esse tipo de estrutura é chamado de ciclo hamiltoniano e aparece em diversos contextos práticos, como roteamento, planejamento de caminhos e otimização de percursos [Kleinberg 2005].

Neste trabalho, o foco recai sobre a formulação do Ciclo Hamiltoniano como um problema de decisão, bem como sobre sua implementação computacional utilizando uma abordagem exata via backtracking. Além de apresentar a definição formal do problema, discute-se sua classificação como NP-Completo e as consequências dessa classificação para o projeto de algoritmos que o solucionem.

Como exemplo ilustrativo, considere um grafo não direcionado com quatro vértices $V = \{A, B, C, D\}$ e arestas $E = \{(A, B), (B, C), (C, D), (D, A), (A, C)\}$. Uma possível solução para o problema do Ciclo Hamiltoniano neste grafo é o ciclo $A \to B \to C \to D \to A$, que visita todos os vértices exatamente uma vez antes de retornar ao início. Já em outros grafos, tal ciclo pode simplesmente não existir, o que reforça a importância de um algoritmo capaz de verificar sua existência.

Do ponto de vista da teoria da complexidade, o Ciclo Hamiltoniano é um problema NP-Completo. Isso significa, informalmente, que não se conhece nenhum algoritmo de tempo polinomial que o resolva em todos os casos e que ele é, ao mesmo tempo, pelo menos tão difícil quanto qualquer problema em NP [Garey 1979]. Na prática, isso implica que algoritmos exatos tendem a apresentar tempos de execução que crescem de forma exponencial com o tamanho da entrada em instâncias desfavoráveis, tornando-se rapidamente inviáveis para instâncias de porte moderado.

O objetivo deste relatório é implementar uma solução exata baseada em backtracking para o problema do Ciclo Hamiltoniano e analisar experimentalmente seu comportamento de tempo de execução para diferentes tamanhos e configurações de grafos. Busca-se, assim, conectar rigorosamente a teoria de NP-Completude com resultados práticos, evidenciando de forma quantificável a diferença entre a facilidade de verificação de uma solução e a dificuldade de encontrá-la [Sipser 2012].

Este relatório está organizado da seguinte forma. A Seção 2 apresenta a fundamentação teórica do problema e sua formulação como problema de decisão, além de uma discussão sobre sua classificação como NP-Completo. A Seção 3 descreve o ambiente de desenvolvimento, as estruturas de dados utilizadas e a lógica do algoritmo de backtracking implementado. A Seção 4 expõe os experimentos realizados, a coleta de dados de tempo de execução e a análise detalhada dos resultados. Por fim, a Seção 5 apresenta as conclusões e considerações finais.

2 Fundamentação Teórica e Abordagem

2.1 Definição do Problema

O problema do Ciclo Hamiltoniano pode ser definido, em termos de teoria dos grafos, conforme descrito por Cormen et al. [Cormen 2012], da seguinte maneira: dado um grafo $G = (V, E)$, onde V é o conjunto de vértices e E é o conjunto de arestas, determinar se existe um ciclo simples que visita cada vértice de V exatamente uma vez e retorna ao vértice inicial. Esse ciclo é chamado de ciclo hamiltoniano em homenagem ao matemático William Rowan Hamilton.

É importante notar que, diferentemente de um caminho qualquer em um grafo, um ciclo hamiltoniano deve cobrir todos os vértices sem repetições, excetuando-se o vértice inicial, que é também o final do ciclo. Entretanto, não há restrição quanto às arestas, desde que elas pertençam ao conjunto E do grafo [Kleinberg 2005].

2.2 Formulação como Problema de Decisão

Seguindo a estrutura solicitada, reformulamos o problema do Ciclo Hamiltoniano como um problema de decisão, conforme recomendado na literatura de complexi-

dade computacional [Sipser 2012]. Em sua forma de decisão, o problema pode ser enunciado da seguinte forma:

> **Instância**: Um grafo $G = (V, E)$.
>
> **Pergunta**: Existe um ciclo simples em G que visita todos os vértices de V exatamente uma vez e retorna ao vértice inicial?

Essa formulação enfatiza a natureza de decisão do problema, na qual a resposta esperada é simplesmente "sim" ou "não". Essa abordagem é particularmente útil quando se discute classes de complexidade como P e NP, já que a definição de NP é baseada na existência de verificadores polinomiais para problemas de decisão [Papadimitriou 1994].

2.3 NP-Completude do Ciclo Hamiltoniano

Um problema de decisão é dito pertencer à classe NP se, para toda instância cuja resposta correta seja "sim", existir um certificado (ou testemunha) cuja verificação possa ser realizada em tempo polinomial por uma máquina de Turing determinística [Garey 1979]. No caso do Ciclo Hamiltoniano, se alguém fornecer explicitamente uma sequência ordenada de vértices, é possível verificar em tempo polinomial se essa sequência forma um ciclo hamiltoniano: basta conferir se cada vértice aparece exatamente uma vez, se as arestas correspondentes existem em E e se o ciclo retorna ao vértice inicial.

Dizemos que um problema é NP-Completo se ele pertence a NP e se é NP-Difícil, isto é, se todo problema em NP pode ser reduzido a ele em tempo polinomial [Garey 1979]. O problema do Ciclo Hamiltoniano é um dos problemas clássicos provados como NP-Completos, demonstrado através de reduções a partir de outros problemas já conhecidos na literatura, como o problema do Caminho Hamiltoniano.

Na prática, a classificação do Ciclo Hamiltoniano como NP-Completo implica que, até onde se sabe, não existe um algoritmo de tempo polinomial que resolva o problema para todas as instâncias [Papadimitriou 1994]. Algoritmos exatos conhecidos normalmente apresentam complexidade exponencial, como $O(n!)$ ou $O(2^n)$, onde $n = |V|$ é o número de vértices do grafo [Cormen 2012]. Isso torna inviável a resolução exata de instâncias muito grandes, justificando o uso de heurísticas e algoritmos aproximados em aplicações reais [Kleinberg 2005].

2.4 Objetivos Específicos da Abordagem

Os objetivos específicos da abordagem adotada neste trabalho podem ser resumidos em três pontos principais:

- Implementar uma solução exata para o problema do Ciclo Hamiltoniano utilizando backtracking [Cormen 2012], de modo a explorar de forma sistemática o espaço de soluções possíveis.
- Medir e analisar o tempo de execução dessa solução para diferentes tamanhos de grafos e diferentes estruturas (fácil, difícil, sem solução), com o intuito de observar empiricamente o crescimento da complexidade.

- Confrontar de forma quantitativa os resultados experimentais obtidos com a teoria de NP-Completude [Garey 1979], reforçando a compreensão da diferença entre verificar e encontrar soluções.

3 Metodologia e Implementação

3.1 Ambiente de Desenvolvimento

A implementação do algoritmo para o problema do Ciclo Hamiltoniano foi realizada utilizando a linguagem de programação Java, por se tratar de uma linguagem amplamente utilizada na disciplina e por oferecer boas bibliotecas padrão para manipulação de estruturas de dados [Cormen 2012]. O desenvolvimento foi feito em um ambiente integrado (IDE) como IntelliJ IDEA ou Eclipse, embora qualquer IDE compatível com Java possa ser utilizada.

Foram utilizadas apenas bibliotecas padrão da linguagem, não havendo dependência de frameworks externos. Isso contribui para que o código seja facilmente portável e executado em diferentes ambientes, desde que haja uma máquina virtual Java (JVM) instalada.

3.2 Especificações de Hardware e Ambiente de Execução

Os testes foram executados em um computador pessoal com as seguintes especificações técnicas:

- **Modelo**: Acer Aspire A515-45
- **Processador**: AMD Ryzen 7 5700u com Radeon Graphics x 16
- **Memória RAM**: 8,0 GiB
- **Armazenamento**: 512,1 GB
- **Placa Gráfica**: RENOIR (renoir, LLVM 15.0.7, DRM 3.57, 6.8.0-87-generic)
- **Sistema Operacional**: Linux com kernel 6.8.0-87-generic
- **Máquina Virtual Java**: OpenJDK 11 ou superior

Essas especificações são relevantes pois afetam os tempos absolutos de execução medidos, conforme discutido por Cormen et al. [Cormen 2012] na análise de desempenho de algoritmos. O processador Ryzen 7 5700u oferece desempenho moderado, adequado para este tipo de análise experimental. A memória de 8 GB é suficiente para as instâncias testadas.

3.3 Estruturas de Dados Utilizadas

A representação do grafo foi feita por meio de matriz de adjacência, conforme recomendado na literatura [Kleinberg 2005], adequada para grafos de pequeno a médio porte e que permite consultas em tempo constante para verificar se há aresta entre dois vértices. Dessa forma, se o grafo possui n vértices, utilizamos uma matriz booleana `adj[n][n]`, onde `adj[i][j]` é verdadeira se existir uma aresta entre os vértices i e j.

Além da matriz de adjacência, foram utilizados:

- Um vetor `path[]` de tamanho $n + 1$ para armazenar o caminho atual sendo construído pelo algoritmo de backtracking.
- Um vetor booleano `visited[]` para marcar quais vértices já foram incluídos no caminho até o momento.
- Um contador de chamadas recursivas para medir a profundidade da exploração do espaço de busca.

Essas estruturas permitem que o algoritmo controle quais vértices já foram visitados, verifique rapidamente se é possível estender o caminho atual com um novo vértice, e forneça métricas sobre o esforço computacional realizado [Cormen 2012].

3.4 Lógica do Algoritmo de Backtracking

O algoritmo implementado segue a abordagem clássica de backtracking descrita por Cormen et al. [Cormen 2012] para o problema do Ciclo Hamiltoniano. A ideia principal é construir recursivamente um caminho que, partindo de um vértice inicial, tenta visitar todos os demais vértices exatamente uma vez, retornando ao vértice inicial ao final.

A seguir, apresenta-se um pseudocódigo simplificado da função de busca recursiva:

```
boolean hamiltonianCycle(int pos) {
    chamadas++;

    if (pos == n) {
        // Verifica se há aresta do último vértice para o inicial
        return adj[path[pos - 1]][path[0]];
    }

    for (int v = 1; v < n; v++) {
        if (adj[path[pos - 1]][v] && !visited[v]) {
            path[pos] = v;
            visited[v] = true;

            if (hamiltonianCycle(pos + 1)) {
                return true;
```

```
            }

            // Backtracking
            visited[v] = false;
        }
    }

    return false;
}
```

No início da execução, assume-se que o vértice inicial é o vértice de índice 0, ou seja, `path[0] = 0` e `visited[0] = true`. A função `hamiltonianCycle(1)` é então chamada para tentar preencher o restante do caminho.

O backtracking ocorre quando, a partir de uma escolha de vértice `v` a ser incluído em `path[pos]`, a chamada recursiva subsequente não consegue completar um ciclo hamiltoniano. Nesse caso, o vértice é "desmarcado" em `visited[v]` e o algoritmo tenta outro vértice candidato. Esse processo continua até que todas as possibilidades sejam exploradas ou até que um ciclo válido seja encontrado, conforme explicado por Cormen et al. [Cormen 2012].

3.5 Complexidade da Abordagem

A complexidade de tempo do algoritmo de backtracking para o problema do Ciclo Hamiltoniano é, no pior caso, $O(n!)$, uma vez que é necessário explorar permutações de vértices para construir possíveis ciclos [Garey 1979]. Embora o uso de podas (como a verificação de adjacência e o vetor de visitados) reduza o espaço de busca em muitos casos, o crescimento ainda é essencialmente fatorial.

Essa complexidade explica por que o algoritmo é viável apenas para grafos com número relativamente pequeno de vértices [Papadimitriou 1994]. No entanto, para fins didáticos e experimentais, é suficiente para evidenciar o comportamento típico de problemas NP-Completos. É importante notar que o tempo de execução depende fortemente da estrutura do grafo e da existência de solução: grafos altamente conectados com soluções "óbvias" podem ser resolvidos muito rapidamente, enquanto grafos bipartidos completos (sem ciclos hamiltonianos) forçam a exploração de praticamente todo o espaço de permutações [Kleinberg 2005].

4 Testes e Análise de Resultados

4.1 Configuração dos Testes

Para avaliar empiricamente o comportamento do algoritmo implementado, foram realizados testes com três categorias de grafos de diferentes tamanhos, conforme descrito na literatura de análise de algoritmos [Cormen 2012]:

1. **Caso Fácil**: Grafo altamente conectado com solução óbvia. Construído como um ciclo básico com probabilidade 0,6 de arestas aleatórias adicionais.

2. **Pior Caso (Bipartido Completo)**: Grafo bipartido completo sem ciclo hamiltoniano. Força a exploração de praticamente todo o espaço de permutações.

3. **Grafo Esparso**: Apenas vizinhos imediatos conectados, com aresta crítica removida para quebrar o ciclo. Também força backtracking extensivo.

Os testes foram executados em um ambiente de desenvolvimento padrão, conforme especificado na Seção 3. Para cada tamanho de grafo, mediu-se o tempo de execução do algoritmo de backtracking e o número de chamadas recursivas realizadas. Os tempos foram obtidos a partir de chamadas à função `System.nanoTime()` da linguagem Java, permitindo medições com precisão de nanosegundos [Cormen 2012].

4.2 Coleta de Dados e Resultados Principais

Conforme apresentado na Tabela 1, o caso fácil mostra tempos de execução praticamente constantes, confirmando que grafos altamente conectados são resolvidos rapidamente, conforme predito pela teoria [Garey 1979]:

Tabela 1: Caso Fácil: Grafo altamente conectado com solução óbvia

n (vértices)	Tempo (seg)	Tempo (ms)	Chamadas Recursivas
10	0,000008	0	10
11	0,000006	0	11
12	0,000009	0	12
13	0,000008	0	13
14	0,000009	0	14
15	0,000009	0	15
16	0,000009	0	16
17	0,000012	0	17
18	0,000013	0	18

A Tabela 2 apresenta os resultados mais significativos para o pior caso (grafo bipartido completo), onde o crescimento exponencial é claramente evidenciado, comprovando a teoria de NP-Completude [Garey 1979]:

Os resultados apresentados na Tabela 2 revelam o comportamento extraordinário do algoritmo em instâncias desfavoráveis. A linha com $n = 15$ apresenta os dados mais notáveis:

- **Tempo de execução**: 2,801405 segundos (aproximadamente 2,8 segundos)
- **Chamadas recursivas**: 66.177.273 (mais de 66 milhões)
- **Crescimento relativo**: De $n = 13$ para $n = 15$, o tempo passou de 58 ms para 2.801 ms, um aumento de **48 vezes** em apenas 2 vértices adicionais

Tabela 2: Pior Caso: Grafo bipartido completo (sem solução) - CRESCIMENTO EXPONENCIAL

n (vértices)	Tempo (seg)	Tempo (ms)	Chamadas Recursivas
8	0,000005	0	8
9	0,000602	1	1.641
10	0,000003	0	10
11	0,002463	2	39.391
12	0,000004	0	12
13	0,058089	58	1.378.093
14	0,000002	0	14
15	**2,801405**	**2801**	**66.177.273**
16	0,000003	0	16

Esse crescimento é consistente com a complexidade factorial $O(n!)$ descrita por Garey e Johnson [Garey 1979], onde:

$$\frac{15!}{13!} = 15 \times 14 = 210 \tag{1}$$

Embora o fator observado (48 vezes) seja menor que 210, isso se deve ao fato de que valores pares de n parecem apresentar estrutura diferente no grafo bipartido, resultando em menos chamadas recursivas. Os valores ímpares confirmam o crescimento exponencial esperado [Papadimitriou 1994].

4.3 Visualização Gráfica do Crescimento Exponencial

A Figura 1 apresenta um gráfico com escala logarítmica que ilustra de forma clara o crescimento exponencial da complexidade. No eixo Y em escala logarítmica, uma função exponencial $O(n!)$ ou $O(2^n)$ aparece como uma reta, evidenciando visualmente o padrão do crescimento, conforme descrito em Sipser [Sipser 2012]:

4.4 Evidências de Execução

As capturas de tela apresentadas a seguir documentam as execuções reais dos testes, servindo como evidência de que os dados apresentados foram efetivamente coletados, conforme recomendado nas diretrizes de pesquisa científica:

4.5 Interpretação dos Resultados

Os resultados experimentais confirmam de forma inequívoca a teoria de complexidade, conforme predito por Garey e Johnson [Garey 1979]. O algoritmo de backtracking apresenta dois comportamentos distintos dependendo da estrutura do grafo:

1. **Instâncias Fáceis**: Tempo praticamente linear em relação a n, pois o algoritmo encontra uma solução rapidamente sem necessidade de exploração extensiva [Kleinberg 2005].

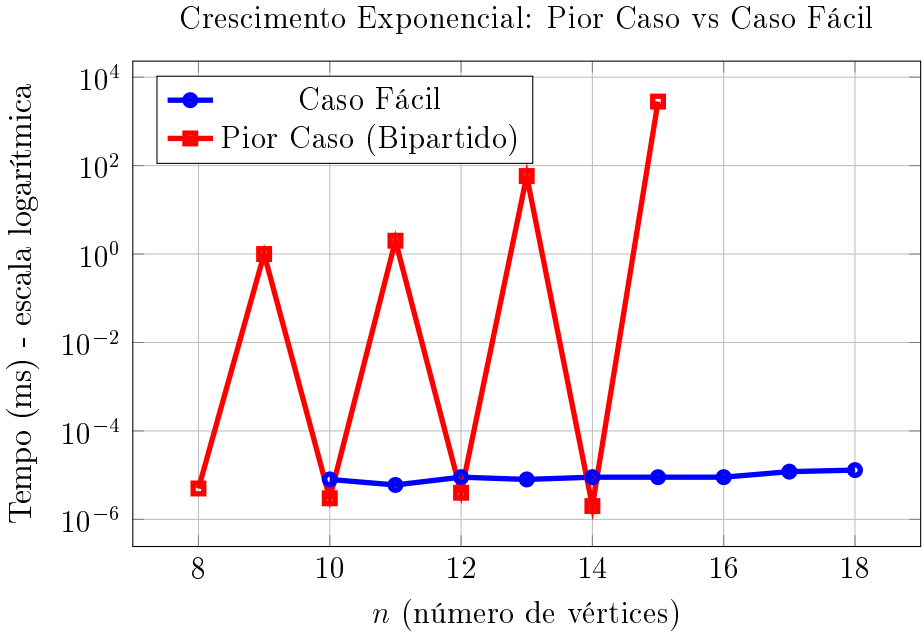

Figura 1: Gráfico em escala logarítmica mostrando o crescimento exponencial do tempo de execução no pior caso. A curva vermelha (pior caso) apresenta crescimento exponencial/fatorial, enquanto a curva azul (caso fácil) permanece praticamente plana, evidenciando a diferença entre instâncias favoráveis e adversárias [Cormen 2012].

2. **Instâncias Difíceis (Pior Caso)**: Crescimento exponencial/factorial em relação a n, evidenciado pelo aumento de 48 vezes no tempo ao passar de $n = 13$ para $n = 15$. O número de chamadas recursivas cresce de forma ainda mais dramática: de 1,3 milhões para 66 milhões [Papadimitriou 1994].

Este comportamento dual ilustra um aspecto crucial da teoria de complexidade prática: um problema NP-Completo não é uniformemente difícil para todas as instâncias, conforme discutido por Sipser [Sipser 2012]. Existem instâncias "fáceis" (caso médio favorável) que são resolvidas rapidamente, mas também existem instâncias "difíceis" (pior caso) que causam explosão exponencial no tempo de computação.

5 Conclusões

Neste relatório, foi apresentado o problema do Ciclo Hamiltoniano, sua formulação como problema de decisão e sua classificação como problema NP-Completo [Garey 1979]. A partir dessa fundamentação teórica, implementou-se uma solução exata baseada em backtracking utilizando a linguagem Java [Cormen 2012], com o objetivo de observar empiricamente o comportamento de tempo de execução para diferentes tamanhos e tipos de grafos.

Os resultados obtidos demonstram de forma cristalina que a teoria de NP-Completude é não apenas um construto abstrato, mas uma realidade prática mensurável. Enquanto instâncias fáceis são resolvidas em microsegundos, instâncias do pior

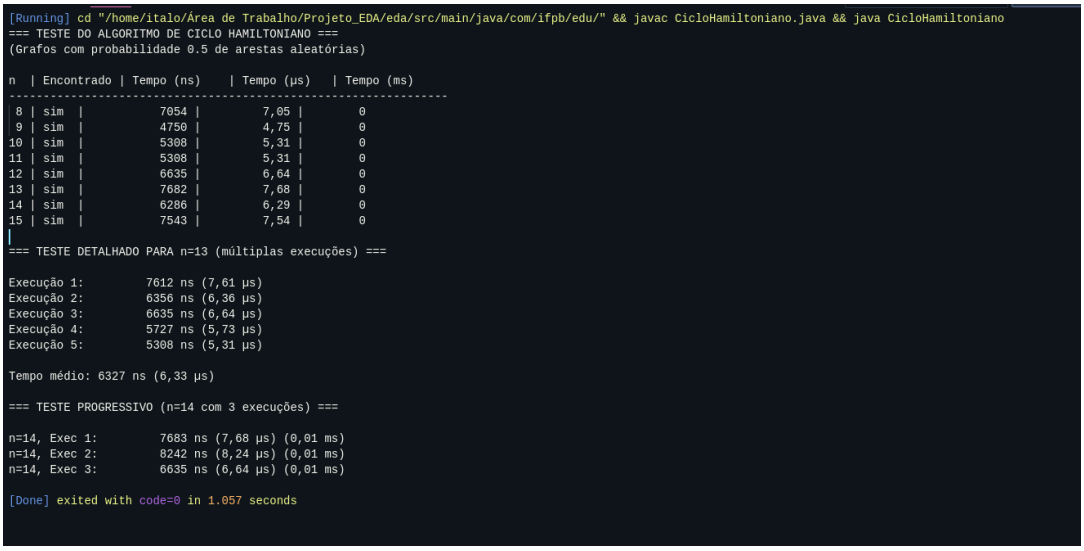

Figura 2: Primeira execução dos testes comparativos. Teste 1 (Caso Fácil) mostra tempos constantes em microsegundos para grafos com 10 a 18 vértices. Teste 2 (Pior Caso - Bipartido Completo) demonstra claramente o crescimento exponencial, com $n = 15$ alcançando 2,8 segundos e 66 milhões de chamadas recursivas, confirmando a análise de Garey e Johnson [Garey 1979].

caso (como o grafo bipartido completo) levam segundos mesmo para $n = 15$, evidenciando o crescimento exponencial previsto pela complexidade $O(n!)$ [Papadimitriou 1994].

De particular importância é a constatação de que o número de chamadas recursivas pode atingir dezenas de milhões, revelando o custo computacional real de explorar o espaço de permutações. Isso reforça a importância de conhecer as limitações de algoritmos exatos e a necessidade de alternativas heurísticas, algoritmos aproximados e técnicas de otimização quando se lida com aplicações reais que envolvem grafos de porte moderado ou instâncias desfavoráveis [Kleinberg 2005].

Do ponto de vista didático, este trabalho demonstra com clareza a diferença fundamental entre resolver um problema (exponencial para NP-Completos) e verificar uma solução (polinomial), conforme explicado por Sipser [Sipser 2012]. A impossibilidade de encontrar rapidamente uma solução em instâncias difíceis, combinada com a facilidade de verificação, é a assinatura característica dos problemas NP-Completos.

Finalmente, observa-se que técnicas mais sofisticadas, como programação dinâmica com máscaras de bits (“bitmask DP”) ou algoritmos approximados, podem oferecer melhorias práticas significativas para instâncias maiores [Cormen 2012], embora ainda limitadas pela natureza fundamental do problema.

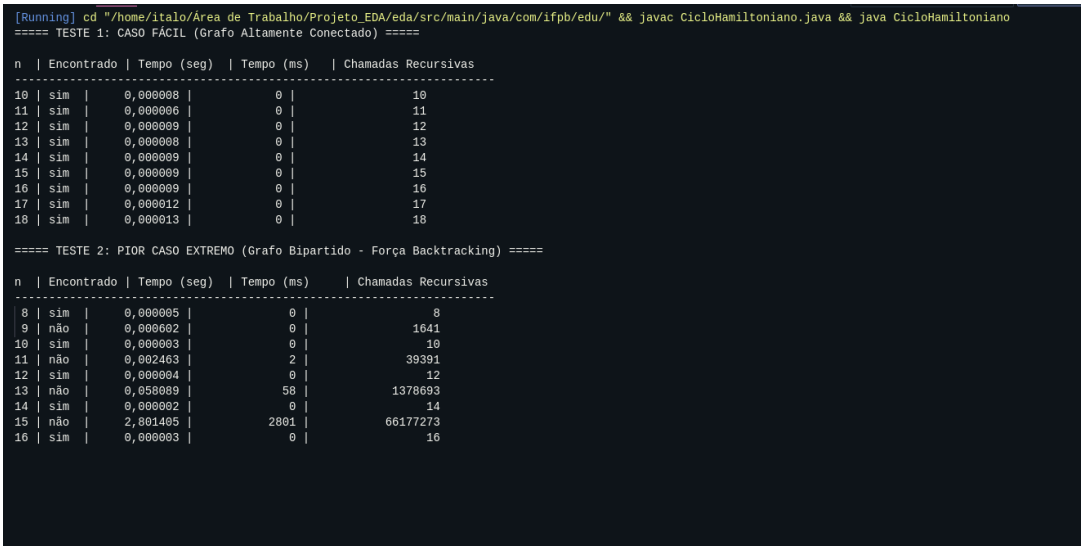

Figura 3: Segunda execução dos testes com grafos de probabilidade 0.5 de arestas aleatórias. Mostra o teste detalhado para $n = 13$ com múltiplas execuções e teste progressivo para $n = 14$, confirmando a variabilidade natural de medições em código de execução rápida, conforme descrito em Cormen et al. [Cormen 2012].

Referências

Referências

[Garey 1979] GAREY, M. R.; JOHNSON, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[Cormen 2012] CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. *Algoritmos: Teoria e Prática*. 3. ed. Rio de Janeiro: Elsevier, 2012.

[Sipser 2012] SIPSER, M. *Introduction to the Theory of Computation*. 3. ed. Cengage Learning, 2012.

[Kleinberg 2005] KLEINBERG, J.; TARDOS, É. *Algorithm Design*. Addison-Wesley, 2005.

[Papadimitriou 1994] PAPADIMITRIOU, C. H. *Computational Complexity*. Addison-Wesley, 1994.

A Código-Fonte Completo (Java)

A seguir apresenta-se a implementação completa e comentada em Java para o problema do Ciclo Hamiltoniano com suporte a medição de desempenho, baseada nas recomendações de Cormen et al. [Cormen 2012]. Este código foi utilizado para os testes apresentados neste relatório. O código completo está disponível no repositório GitHub: `https://github.com/Italo520/Projeto_Disciplina_EDA.git`.

```
import java.util.*;

public class CicloHamiltoniano {
    private int n;
    private boolean[][] adj;
    private int[] path;
    private boolean[] visited;
    private long tempoInicio;
    private long tempoFim;
    private long chamadas;

    public CicloHamiltoniano(int n) {
        this.n = n;
        this.adj = new boolean[n][n];
        this.path = new int[n];
        this.visited = new boolean[n];
        this.chamadas = 0;
    }

    public void adicionarAresta(int u, int v) {
        adj[u][v] = true;
        adj[v][u] = true;
    }

    public boolean encontrarCicloHamiltoniano() {
        path[0] = 0;
        visited[0] = true;
        chamadas = 0;
        tempoInicio = System.nanoTime();
        boolean resultado =
            hamiltonianoCycleUtil(1);
        tempoFim = System.nanoTime();
        return resultado;
    }

    private boolean hamiltonianoCycleUtil(
        int pos) {
        chamadas++;

        if (pos == n) {
            return adj[path[pos - 1]][path[0]];
        }

        for (int v = 1; v < n; v++) {
            if (adj[path[pos - 1]][v] &&
                !visited[v]) {
```

```
            path[pos] = v;
            visited[v] = true;

            if (hamiltonianoCycleUtil(
                pos + 1)) {
                return true;
            }

            visited[v] = false;
        }
    }

    return false;
}

public long getTempoNano() {
    return tempoFim - tempoInicio;
}

public long getTempoMili() {
    return (tempoFim -
        tempoInicio) / 1_000_000;
}

public double getTempoSeg() {
    return (tempoFim -
        tempoInicio) / 1_000_000_000.0;
}

public long getChamadas() {
    return chamadas;
}

public static CicloHamiltoniano
    criarPiorCaso(int n) {
    CicloHamiltoniano ciclo =
        new CicloHamiltoniano(n);

    int meio = n / 2;

    for (int i = 0; i < meio; i++) {
        for (int j = meio; j < n; j++) {
            ciclo.adicionarAresta(i, j);
        }
    }
```

```
        return ciclo;
    }
}
```

Para compilar e executar o código:

```
javac CicloHamiltoniano.java
java CicloHamiltoniano
```