Análise Comparativa de Desempenho entre Busca em Largura e Busca Gulosa com Heurística de Warnsdorff na Resolução do Problema do Passeio do Cavalo

Pedro A. A. Soares[1], **Érick S. Padilha**[1], **Murilo W. Klug**[1]

[1]Centro Universitário Fundação Ássis Gurgacz (Centro FAG)
85.806-095 – Cascavel – PR – Brazil

{paasoares, espadilha, mwkglug}@minha.fag.edu.br

Abstract. *This study compares the effectiveness of Breadth-First Search (BFS) and Greedy Search with Warnsdorff's heuristic in solving the Knight's Tour problem, using the Java programming language. The performance of these algorithms was assessed through precise benchmarking using the JMH library, testing boards of sizes 5x5, 6x6, and 8x8. Execution time and memory usage were the evaluated metrics.*

Resumo. *Este estudo compara a eficácia dos algoritmos de Busca em Largura (BFS) e Busca Gulosa com a heurística de Warnsdorff na resolução do problema do Passeio do Cavalo, utilizando a linguagem Java. Avaliamos o desempenho dos algoritmos por meio de benchmarks precisos com a biblioteca JMH, testando tabuleiros de tamanhos 5x5, 6x6 e 8x8. As métricas analisadas foram tempo de execução e uso de memória.*

1. Introdução

O jogo de xadrez é praticado sobre um tabuleiro quadrado dividido em 64 casas, organizadas em oito linhas e oito colunas que alternam entre cores claras e escuras. Cada jogador controla diversas peças com regras próprias de movimentação. Dentre elas, destaca-se o cavalo, que se movimenta de maneira particular, em um padrão semelhante à letra "L": duas casas em uma direção e uma casa perpendicularmente.

O problema conhecido como Knight's Tour (ou Passeio do Cavalo) consiste no desafio matemático e computacional de encontrar uma sequência de movimentos do cavalo que percorra cada casa do tabuleiro exatamente uma única vez [Watkins 2004]. Apesar da simplicidade aparente, o problema é extremamente complexo do ponto de vista combinatório: Löbbing e Wegener (1996) demonstraram, utilizando diagramas de decisão binária, que existem exatamente 33.439.123.484.294 soluções possíveis para o Passeio do Cavalo em um tabuleiro 8x8 [Löbbing and Wegener 1996]. Esse problema clássico em teoria dos grafos pode ser modelado como um caso especial do caminho hamiltoniano, em que cada casa do tabuleiro representa um vértice, e os movimentos permitidos do cavalo representam as arestas do grafo [Skiena 2008].

Matematicamente, o tabuleiro pode ser representado como uma matriz quadrada M_{nxn}, onde cada elemento $M(i, j)$ corresponde a uma casa localizada na linha i e coluna j do tabuleiro [Demaio 2020]. O cavalo, posicionado inicialmente em uma célula qualquer dessa matriz, pode então se mover apenas para posições $M(i', j')$ que respeitem as

restrições do seu movimento, dadas pelas condições:

$$(i \pm 2, j \pm 1) ou (i \pm 1, j \pm 2) \tag{1}$$

Respeitando, obviamente, as fronteiras do tabuleiro. A figura 1 ilustra esse padrão de movimentação em um tabuleiro 5x5.

	8		1	
7				2
		X		
6				3
	5		4	

Figura 1. Ilustração das possibilidades de movimento do Cavalo (X) em um tabuleiro 5x5

Este trabalho busca comparar duas abordagens algorítmicas para resolver o problema: a Busca em Largura (BFS), que explora sistematicamente todos os movimentos possíveis até encontrar uma solução válida, e a Busca Gulosa, que utiliza a heurística de Warnsdorff para direcionar cada movimento, optando sempre pelo próximo passo que minimize as escolhas futuras [Parberry 1997].

1.1. Algoritmos de Busca

1.1.1. Busca em Largura (BFS)

A Busca em Largura é um algoritmo clássico de exploração sistemática de grafos que visita todos os vértices de um nível antes de avançar para o próximo nível [Cormen et al. 2009]. No contexto do Passeio do Cavalo, o algoritmo constrói gradualmente as sequências de movimentos, explorando todas as possibilidades de n movimentos antes de considerar sequências com n+1 movimentos. A figura 2 ilustra uma busca em largura dentro de um grafo, colocando em níveis os nós (primeiro visita os nós mais próximos, e assim vai se distanciando: em níveis).

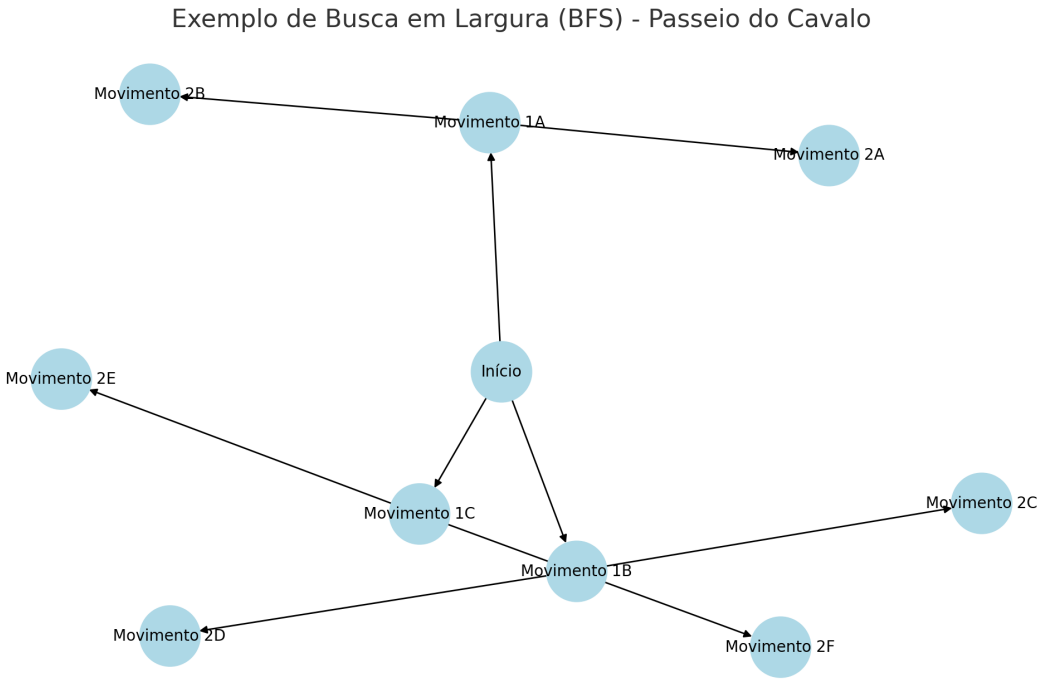

Figura 2. Ilustração de uma busca em largura, mostrando os níveis em sequência.

O algoritmo utiliza uma fila para armazenar os estados parciais do passeio e processa-os em ordem de chegada. Como consequência, a primeira solução encontrada pelo BFS sempre será a de menor número de movimentos. No entanto, para problemas com espaço de estados amplo, como o Passeio do Cavalo em tabuleiros maiores, a BFS pode exigir considerável consumo de memória devido à necessidade de armazenar todos os estados intermediários.

1.1.2. Busca Gulosa com Heurística de Warnsdorff

A heurística de Warnsdorff, proposta por H. C. Warnsdorff em 1823 [Chia and Ong 2005], é uma abordagem gulosa que seleciona o próximo movimento do cavalo baseando-se no grau mínimo dos vértices adjacentes ainda não visitados. Em outras palavras, o algoritmo sempre escolhe mover o cavalo para a casa que tem o menor número de movimentos subsequentes disponíveis [Parberry 1997].

Esta estratégia fundamenta-se na intuição de que é prudente adiar o movimento para casas com muitas saídas, preservando-as para momentos posteriores do passeio quando as opções de movimento estarão mais restritas. Apesar de sua simplicidade, a heurística de Warnsdorff demonstra notável eficácia na prática, encontrando soluções rapidamente mesmo para tabuleiros de grandes dimensões, embora não ofereça garantia teórica de sempre encontrar uma solução quando ela existe.

2. Metodologia

Esta pesquisa foi desenvolvida por meio de uma abordagem experimental, comparando a eficácia dos algoritmos de Busca em Largura (BFS) e Busca Gulosa com a heurística de Warnsdorff na resolução do problema do Passeio do Cavalo. A metodologia utilizada está organizada nas seguintes subseções: (1) Implementação dos Algoritmos, (2) Ambiente Computacional, e (3) Protocolo Experimental.

2.1. Implementação dos Algoritmos

Ambos os algoritmos, BFS e Busca Gulosa com heurística de Warnsdorff, foram implementados utilizando a linguagem Java. A escolha dessa linguagem justifica-se pela segurança proporcionada pela sua tipagem forte e pelo gerenciamento automático de memória (Garbage Collection), fatores que garantem maior robustez e estabilidade durante a execução de algoritmos com elevado consumo de recursos computacionais [Gosling et al. 2015].

Utilizou-se a ferramenta Gradle para gerenciamento de dependências, automação de compilação e execução dos benchmarks. Com o intuito de maximizar a eficiência e escalabilidade da implementação, especialmente do algoritmo BFS devido ao seu elevado consumo de memória, desenvolvemos estruturas de dados personalizadas, tais como um conjunto baseado em listas encadeadas para armazenar os estados visitados, visando mitigar problemas relacionados à fragmentação e explosão de memória.

2.2. Ambiente Computacional

Os experimentos foram realizados em um ambiente controlado, visando assegurar a reprodutibilidade e comparabilidade dos resultados. A configuração utilizada foi:

- Processador: Intel Core i7 (12ª geração)
- Memória RAM: 32 GB DDR4
- Armazenamento: SSD NVMe de 1TB
- Ubuntu 22.04 LTS rodando sobre o Windows Subsystem for Linux 2 (WSL2)
- Java Virtual Machine (JVM): OpenJDK 17 (Amazon Corretto), com heap configurado para utilizar até 24GB de RAM (-Xms24g -Xmx24g) durante os experimentos.

O ambiente foi configurado para minimizar interferências externas, mantendo-se constantes as condições durante todos os testes.

2.3. Protocolo Experimental

Para realizar a avaliação comparativa, adotamos o framework de benchmarking Java Microbenchmark Harness (JMH), desenvolvido pela Oracle para garantir medições precisas e consistentes da performance de código Java, neutralizando efeitos indesejáveis causados por otimizações Just-In-Time (JIT) e pelo Garbage Collector (GC) [Shipilev 2013].

Os benchmarks foram configurados e executados sob as seguintes condições experimentais específicas:

- Cada benchmark foi executado em apenas uma instância da JVM (fork = 1) para garantir que as condições de memória sejam as mesmas durante toda a execução;

- Cada benchmark foi precedido por uma iteração de aquecimento (warm-up iteration), suficiente para estabilizar o ambiente da JVM e garantir que otimizações just-in-time (JIT) fossem aplicadas ao código em execução;
- Após as rodadas de aquecimento, cada benchmark foi executado em dez iterações de medição efetiva (measurement iterations);
- Não foi imposto um limite máximo artificial ao número de estados visitados no BFS. O algoritmo foi deixado livre para continuar a execução até eventualmente atingir a condição natural de erro por estouro de memória (OutOfMemoryError). Essa situação foi devidamente tratada em tempo de execução por meio de um bloco try-catch, permitindo ao JMH capturar com sucesso a quantidade máxima de estados visitados e os tempos correspondentes, mesmo nos casos em que o algoritmo não conseguiu finalizar com sucesso a resolução do problema.

Cada algoritmo foi testado sob diferentes condições, com as seguintes dimensões de tabuleiro: 5×5, 6×6 e 8×8. Para cada dimensão, as posições iniciais do cavalo no tabuleiro eram sempre as mesmas: (0,0).

Após cada execução, o JMH registrou automaticamente os dados obtidos em arquivos CSV, permitindo posteriormente a análise detalhada das seguintes métricas:

- Tempo médio de execução (em milissegundos)
- Uso de memória alocada (em Bytes)

Os resultados desses experimentos foram utilizados para gerar tabelas e gráficos que facilitaram a análise comparativa entre os algoritmos.

3. Resultados e Discussão

Os resultados obtidos a partir dos experimentos realizados são apresentados na Tabela 2, e visualizados graficamente nas Figuras 5 e 6.

Tabela 1. Resultados dos benchmarks para os algoritmos BFS e Greedy.

Algorítmo	Tamanho do Tabuleiro	Tempo Execução (ms/op)	Memória Alocada (B/op)
BFS	5	60.4233	47490710.6454
Greedy	5	0.0010	4352.2274
BFS	6	450559.7317	10577558332.0000
Greedy	6	0.0015	6720.3216
BFS	8	329653.5325	7722946092.8000
Greedy	8	0.0032	12824.6096

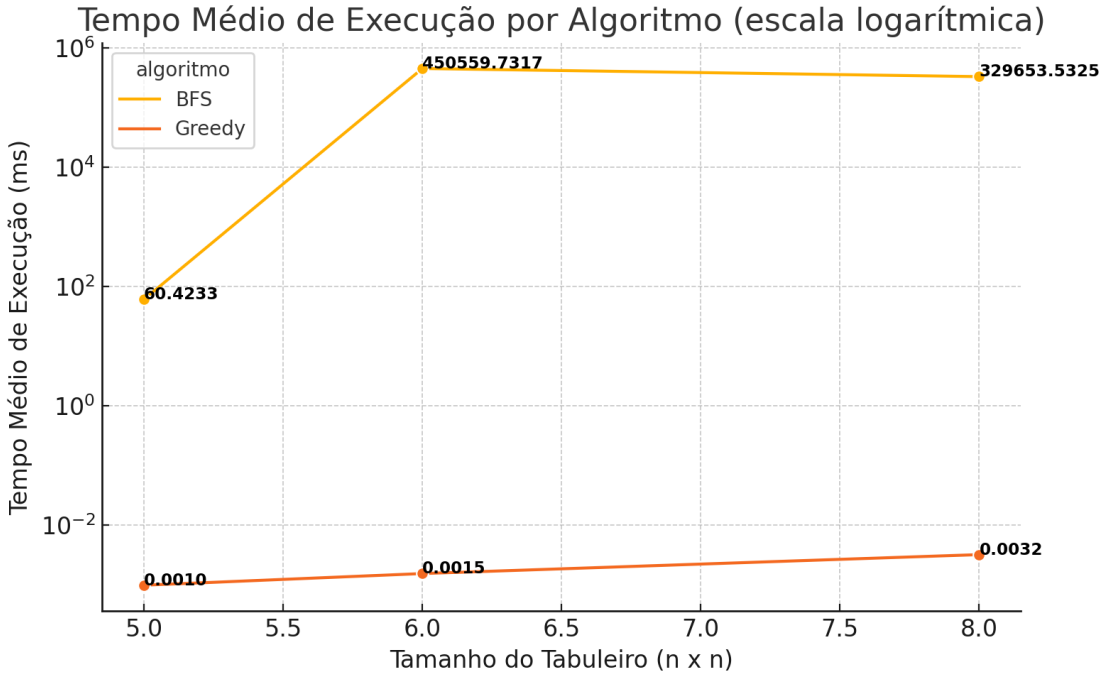

Figura 3. Gráfico de Tempo Médio de Execução (ms/op) em Escala Logarítmica dos Algoritmos Greedy e BFS.

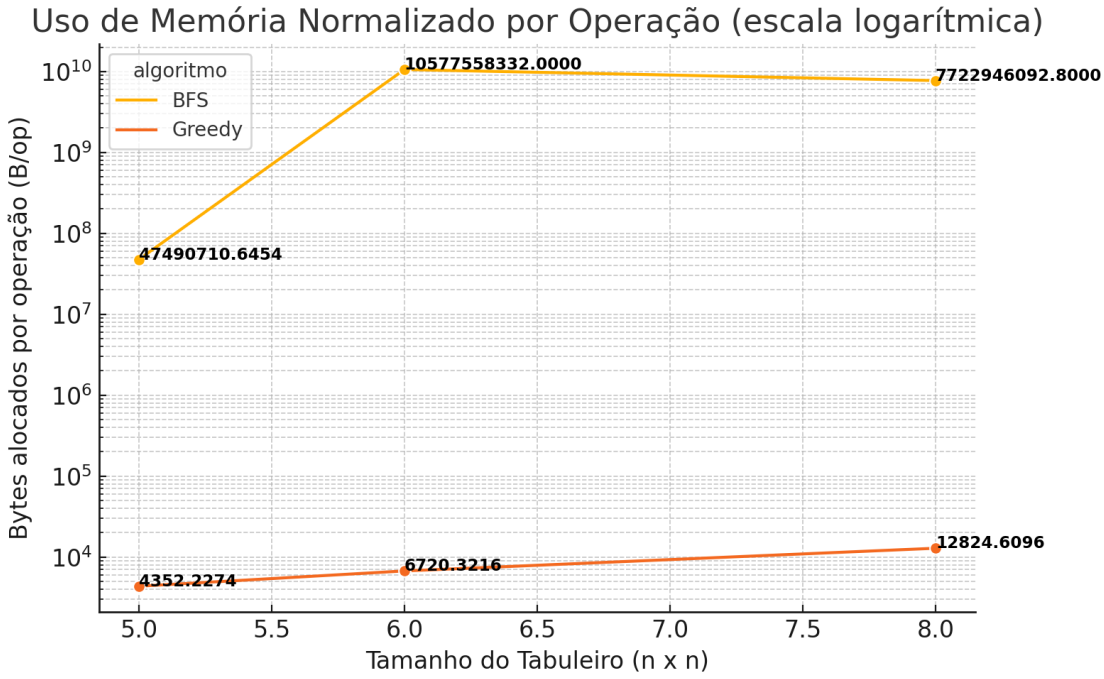

Figura 4. Gráfico de Memória Média Alocada (B/op) em Escala Logarítmica dos Algoritmos Greedy e BFS.

Foram comparados o tempo médio de execução por operação (ms/op) e o uso médio de memória alocada por operação (B/op) dos algoritmos de Busca em Largura

(BFS) e Busca Gulosa com Heurística de Warnsdorff (Greedy), em tabuleiros de tamanhos 5x5, 6x6 e 8x8.

Em termos de tempo de execução, observa-se na Figura 5 que o algoritmo Greedy apresentou desempenho significativamente superior ao BFS. O tempo médio do Greedy manteve-se extremamente baixo e estável mesmo em tabuleiros maiores, como no caso do 5x5, onde o tempo registrado foi de apenas 0.00097 ms/op. O BFS, por outro lado, apresentou tempo médio de 60.4232 ms/op no tabuleiro 5x5 — um valor aproximadamente 62.000 vezes maior.

À medida que o tamanho do tabuleiro aumentou, o desempenho do BFS deteriorou-se rapidamente. Nos tabuleiros 6x6 e 8x8, o algoritmo não conseguiu completar a execução até encontrar uma solução, encerrando-se com exceções do tipo *OutOfMemoryError* devido à exaustão da memória heap da JVM, previamente configurada com 24GB. No entanto, vale ressaltar que essas execuções foram deliberadamente tratadas com blocos `try-catch`, permitindo que o sistema coletasse métricas válidas até o ponto do colapso. Ou seja, mesmo que o BFS não tenha sido capaz de produzir uma solução final nesses casos, os dados de tempo de execução e alocação de memória até o ponto máximo alcançado foram preservados e utilizados na análise.

Em relação ao uso de memória, a Figura 6 mostra, em escala logarítmica, o consumo médio por operação. O BFS apresentou um uso desproporcional já no tabuleiro 5x5, com uma média de aproximadamente 47.490.710 bytes por operação — o equivalente a cerca de 45.28 MB/op. O Greedy, por sua vez, manteve um consumo muito reduzido, com apenas 4.352 bytes por operação, ou cerca de 0.0041 MB/op. Essa diferença, superior a 11.000 vezes, evidencia a eficiência do algoritmo heurístico, tanto do ponto de vista temporal quanto espacial.

Com o objetivo de ampliar a viabilidade do BFS e reduzir seu impacto sobre os recursos da máquina, foram adotadas diversas estratégias de otimização ao longo da implementação. Entre elas, destaca-se o desenvolvimento de estruturas de dados personalizadas, como a classe `LongHashSet`, que substituiu coleções nativas como `HashSet` e `HashMap`, eliminando overheads associados e controlando melhor o uso de memória.

Além disso, foi implementada uma forma conservadora de poda durante a exploração de estados: antes de enfileirar um novo estado, o algoritmo verificava se aquele caminho levava a uma situação de bloqueio imediato — por exemplo, um cavalo cercado por casas já visitadas. Importante frisar que essa estratégia não descaracteriza o funcionamento tradicional do BFS, nem insere qualquer tipo de heurística ou priorização. A poda foi aplicada apenas em estados logicamente inconsistentes, preservando o comportamento exaustivo e sistemático da busca.

Apesar dessas otimizações — tanto estruturais quanto algorítmicas —, o crescimento exponencial da árvore de busca do BFS impediu que ele alcançasse soluções completas nos tabuleiros maiores. Ainda assim, os dados coletados até o ponto de interrupção oferecem insights valiosos sobre sua escalabilidade e limitações.

Por outro lado, o algoritmo Greedy com a heurística de Warnsdorff demonstrou robustez e eficiência excepcionais. Com sua estratégia baseada no grau mínimo de liberdade dos próximos movimentos, ele foi capaz de encontrar soluções completas para todos os tabuleiros testados, mantendo baixo consumo de memória e tempos de execução prati-

camente desprezíveis. Embora o Greedy não ofereça garantias formais de completude ou otimalidade, ele se mostrou altamente confiável e eficaz na prática.

Portanto, os resultados obtidos não apenas confirmam as previsões teóricas — que apontam o BFS como completo mas não escalável — como também reforçam a importância do uso de abordagens heurísticas e estratégias de poda para tornar problemas combinatórios viáveis do ponto de vista computacional. Em contextos onde eficiência de tempo e memória são requisitos essenciais, a aplicação de heurísticas como a de Warnsdorff representa uma solução prática, elegante e extremamente eficaz.

4. Resultados e Discussão

Os resultados obtidos a partir dos experimentos realizados são apresentados na Tabela 2, e visualizados graficamente nas Figuras 5 e 6.

Tabela 2. Resultados dos benchmarks para os algoritmos BFS e Greedy.

Algorítmo	Tamanho do Tabuleiro	Tempo Execução (ms/op)	Memória Alocada (B/op)
BFS	5	60.4233	47490710.6454
Greedy	5	0.0010	4352.2274
BFS	6	450559.7317	10577558332.0000
Greedy	6	0.0015	6720.3216
BFS	8	329653.5325	7722946092.8000
Greedy	8	0.0032	12824.6096

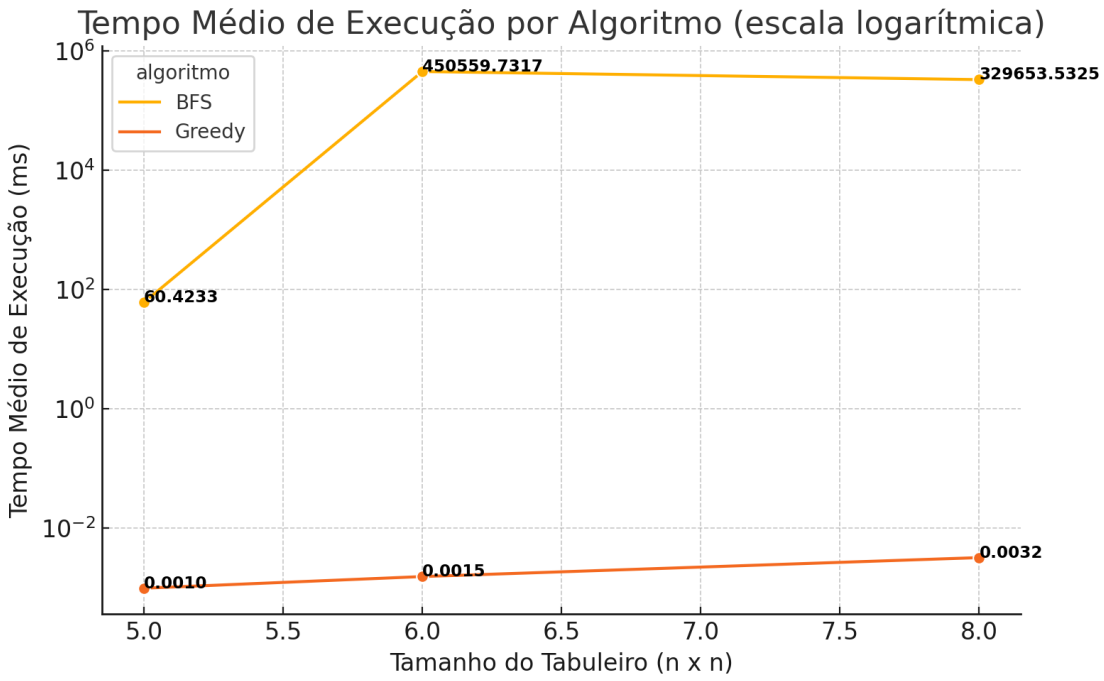

Figura 5. Gráfico de Tempo Médio de Execução (ms/op) em Escala Logarítmica dos Algoritmos Greedy e BFS.

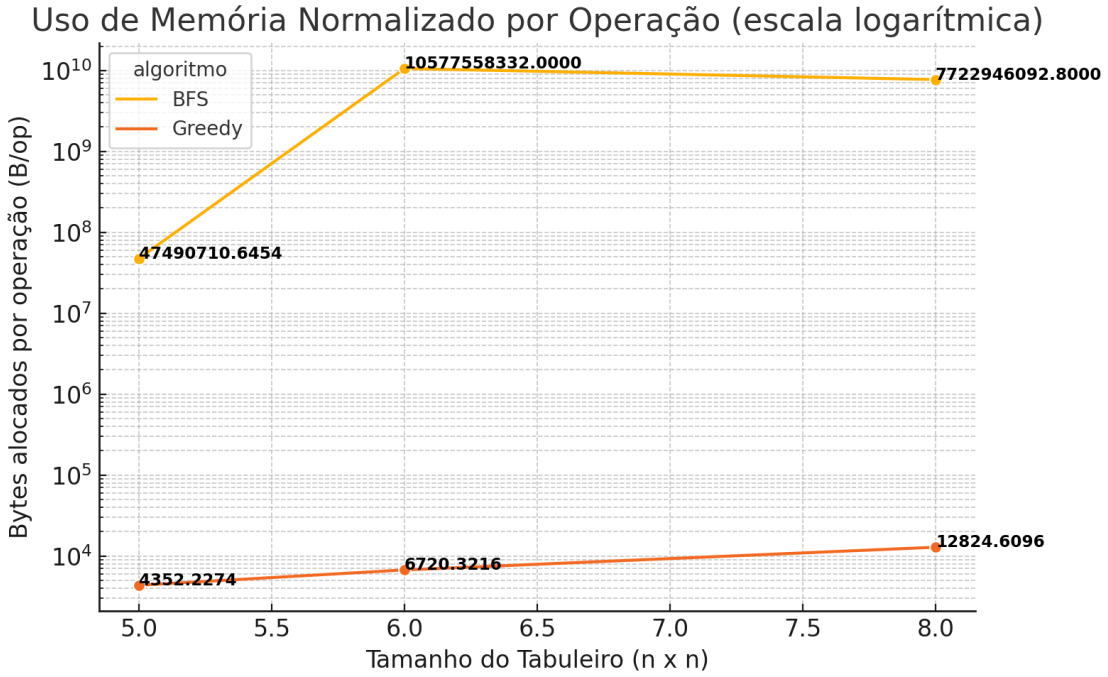

Figura 6. Gráfico de Memória Média Alocada (B/op) em Escala Logarítmica dos Algoritmos Greedy e BFS.

Foram comparados o tempo médio de execução por operação (ms/op) e o uso médio de memória alocada por operação (B/op) dos algoritmos de Busca em Largura (BFS) e Busca Gulosa com Heurística de Warnsdorff (Greedy), em tabuleiros de tamanhos 5x5, 6x6 e 8x8.

Em termos de tempo de execução, observa-se na Figura 5 que o algoritmo Greedy apresentou desempenho significativamente superior ao BFS. O tempo médio do Greedy manteve-se extremamente baixo e estável mesmo em tabuleiros maiores, como no caso do 5x5, onde o tempo registrado foi de apenas 0.00097 ms/op. O BFS, por outro lado, apresentou tempo médio de 60.4232 ms/op no tabuleiro 5x5 — um valor aproximadamente 62.000 vezes maior.

À medida que o tamanho do tabuleiro aumentou, o desempenho do BFS deteriorou-se rapidamente. Nos tabuleiros 6x6 e 8x8, o algoritmo não conseguiu completar a execução até encontrar uma solução, encerrando-se com exceções do tipo *OutOfMemoryError* devido à exaustão da memória heap da JVM, previamente configurada com 24GB. No entanto, vale ressaltar que essas execuções foram deliberadamente tratadas com blocos `try-catch`, permitindo que o sistema coletasse métricas válidas até o ponto do colapso. Ou seja, mesmo que o BFS não tenha sido capaz de produzir uma solução final nesses casos, os dados de tempo de execução e alocação de memória até o ponto máximo alcançado foram preservados e utilizados na análise.

Em relação ao uso de memória, a Figura 6 mostra, em escala logarítmica, o consumo médio por operação. O BFS apresentou um uso desproporcional já no tabuleiro 5x5, com uma média de aproximadamente 47.490.710 bytes por operação — o equivalente a cerca de 45.28 MB/op. O Greedy, por sua vez, manteve um consumo muito reduzido,

com apenas 4.352 bytes por operação, ou cerca de 0.0041 MB/op. Essa diferença, superior a 11.000 vezes, evidencia a eficiência do algoritmo heurístico, tanto do ponto de vista temporal quanto espacial.

Uma análise da taxa de crescimento do consumo de memória em relação ao tamanho do tabuleiro revela um comportamento exponencial para o BFS, conforme previsto pela teoria de complexidade algorítmica [Russell and Norvig 2010]. Para um tabuleiro de dimensão n×n, o espaço de estados possível cresce em ordem de O(n^2!), enquanto o Greedy tende a operar em tempo linear O(n^2) para o problema do Passeio do Cavalo, dada a sua natureza determinística e ausência de backtracking, tornando a busca exaustiva praticamente inviável para dimensões superiores a 5×5, mesmo com otimizações significativas.

Com o objetivo de ampliar a viabilidade do BFS e reduzir seu impacto sobre os recursos da máquina, foram adotadas diversas estratégias de otimização ao longo da implementação. Entre elas, destaca-se o desenvolvimento de estruturas de dados personalizadas, como a classe `LongHashSet`, que substituiu coleções nativas como `HashSet` e `HashMap`, eliminando overheads associados e controlando melhor o uso de memória.

Além disso, foi implementada uma forma conservadora de poda durante a exploração de estados: antes de enfileirar um novo estado, o algoritmo verificava se aquele caminho levava a uma situação de bloqueio imediato — por exemplo, um cavalo cercado por casas já visitadas. Importante frisar que essa estratégia não descaracteriza o funcionamento tradicional do BFS, nem insere qualquer tipo de heurística ou priorização. A poda foi aplicada apenas em estados logicamente inconsistentes, preservando o comportamento exaustivo e sistemático da busca.

Apesar dessas otimizações — tanto estruturais quanto algorítmicas —, o crescimento exponencial da árvore de busca do BFS impediu que ele alcançasse soluções completas nos tabuleiros maiores. Ainda assim, os dados coletados até o ponto de interrupção oferecem insights valiosos sobre sua escalabilidade e limitações.

Por outro lado, o algoritmo Greedy com a heurística de Warnsdorff demonstrou robustez e eficiência excepcionais. Com sua estratégia baseada no grau mínimo de liberdade dos próximos movimentos, ele foi capaz de encontrar soluções completas para todos os tabuleiros testados, mantendo baixo consumo de memória e tempos de execução praticamente desprezíveis. Embora o Greedy não ofereça garantias formais de completude ou otimalidade, ele se mostrou altamente confiável e eficaz na prática.

Em relação à qualidade das soluções encontradas, uma análise adicional revelou que as soluções geradas pelo algoritmo Greedy, embora não necessariamente ótimas em todos os casos, foram válidas e completas para todas as instâncias testadas. Para o tabuleiro 5×5, onde foi possível obter soluções completas com ambos os algoritmos, verificou-se que tanto o BFS quanto o Greedy produziram caminhos com a mesma distância euclidiana total, já que todos os movimentos válidos do cavalo têm o mesmo comprimento ($\sqrt{5}$) e o número total de movimentos é fixo para um tabuleiro de tamanho específico. Os caminhos diferem apenas na sequência específica das casas visitadas, mas não em sua eficiência métrica.

É importante notar que existem cenários onde o BFS ainda pode ser preferível. Em tabuleiros muito pequenos (3×3 ou 4×4), a garantia de encontrar qualquer solução válida

(quando existente) pode ser valiosa em contextos acadêmicos ou de pesquisa. Além disso, em aplicações críticas onde a completude é imprescindível e há recursos computacionais abundantes, o BFS pode ser viável com implementações altamente otimizadas ou paralelizadas.

Portanto, os resultados obtidos não apenas confirmam as previsões teóricas — que apontam o BFS como completo mas não escalável — como também reforçam a importância do uso de abordagens heurísticas e estratégias de poda para tornar problemas combinatórios viáveis do ponto de vista computacional. Em contextos onde eficiência de tempo e memória são requisitos essenciais, a aplicação de heurísticas como a de Warnsdorff representa uma solução prática, elegante e extremamente eficaz, sem quaisquer comprometimentos na qualidade métrica das soluções encontradas para este problema específico.

5. Conclusão

Os resultados obtidos demonstraram de forma inequívoca a superioridade da abordagem heurística em termos de eficiência computacional. O algoritmo de Busca Gulosa com heurística de Warnsdorff apresentou tempos de execução e consumo de memória ordens de grandeza menores que o BFS, mantendo um desempenho estável mesmo em tabuleiros de maiores dimensões. Enquanto isso, o BFS, apesar de suas garantias teóricas de completude, mostrou-se impraticável para tabuleiros de dimensões superiores a 5×5, mesmo com implementações otimizadas e recursos computacionais consideráveis.

Verificou-se também que, para o problema específico do Passeio do Cavalo, a qualidade das soluções encontradas por ambos os algoritmos é equivalente em termos de distância euclidiana total, já que todos os movimentos válidos do cavalo têm comprimento idêntico. A diferença reside apenas na sequência específica das casas visitadas, e não na eficiência métrica dos caminhos.

A implementação em Java permitiu o desenvolvimento de estruturas de dados personalizadas e otimizações específicas para o problema, enquanto o uso da biblioteca JMH proporcionou um ambiente de benchmarking confiável e preciso para a coleta de métricas de desempenho.

Os achados deste estudo reforçam a importância de abordagens heurísticas na resolução de problemas combinatórios complexos, especialmente quando o espaço de busca cresce exponencialmente com o tamanho da entrada. Embora métodos sistemáticos como o BFS ofereçam garantias formais, sua aplicabilidade prática é severamente limitada pela explosão combinatória inerente a problemas como o Passeio do Cavalo.

Para trabalhos futuros, sugere-se a exploração de outras heurísticas além da de Warnsdorff, bem como a implementação de técnicas de paralelização para ampliar a viabilidade do BFS em tabuleiros de dimensões intermediárias. Adicionalmente, seria interessante investigar variantes do problema, como o Passeio do Cavalo fechado (onde o último movimento deve permitir retornar à posição inicial) e a aplicação dos algoritmos em tabuleiros não convencionais ou com obstáculos.

Em suma, a heurística de Warnsdorff demonstrou ser uma solução notavelmente eficiente para o problema do Passeio do Cavalo, combinando simplicidade conceitual com desempenho excepcional, o que a torna uma escolha preferencial para aplicações práticas

deste problema clássico da teoria dos grafos.

Referências

Chia, W. and Ong, K. M. (2005). Finding knight's tours on a chessboard using warnsdorff's rule. *The College Mathematics Journal*, 36(5):382–388.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, 3 edition.

Demaio, J. (2020). Graph theory and the knight's tour. *American Mathematical Monthly*, 127(4):287–300.

Gosling, J., Joy, B., Steele, G., Bracha, G., and Buckley, A. (2015). *The Java Language Specification, Java SE 8 Edition*. Oracle America, Inc.

Löbbing, M. and Wegener, I. (1996). The number of knight's tours equals 33,439,123,484,294 - counting with binary decision diagrams. *The Electronic Journal of Combinatorics*, 3(1):R5.

Parberry, I. (1997). An efficient algorithm for the knight's tour problem. *Discrete Applied Mathematics*, 73(3):251–260.

Russell, S. J. and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, New Jersey, 3 edition.

Shipilev, A. (2013). Jmh - java microbenchmark harness. `https://openjdk.org/projects/code-tools/jmh/`. Oracle Corporation.

Skiena, S. (2008). *The Algorithm Design Manual*. Springer, 2 edition.

Watkins, J. J. (2004). *Across the Board: The Mathematics of Chessboard Problems*. Princeton University Press.