UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Sudoku Solver

Aluno: Lucas de Oliveira Araújo

Matrícula: 333

Belo Horizonte, MG
2024

Sumário

1 Introdução **2**

2 Método **2**

2.1 Busca em espaços de estados 2

2.2 Algoritmos de busca 2

2.3 Grafos e árvores 3

2.4 Implementação 3

2.4.1 Breadth-First Search 4

2.4.2 Iterative Deepening Depth-First Search 4

2.4.3 Uniform-Cost Search 5

2.4.4 A* Search 6

2.4.5 Greedy Best-First Search 6

2.5 Heurísticas 7

2.5.1 Heurística para o A* Search 7

2.5.2 Heurística para o Greedy Best-First Search 8

2.6 Custo de travessia de arestas 8

3 Análise de complexidade **8**

4 Análise Experimental **9**

4.1 Análise comparativa pelo total de estados expandidos 9

4.2 Análise comparativa pelo tempo de execução 10

5 Conclusão **11**

Referências **11**

---

1 Introdução

O Sudoku é um quebra-cabeça baseado na colocação lógica de números na tabuleiro seguindo determinadas regras. Um sudoku padrão é uma matriz 9x9 composta de sub-regiões 3x3. Inicialmente, a matriz pode conter algumas células já preenchidas e o objetivo do jogador é preencher as demais células vazias com números inteiros de 1 a 9, de forma que cada linha, coluna e sub-região deve conter um determinado número exatamente uma vez.

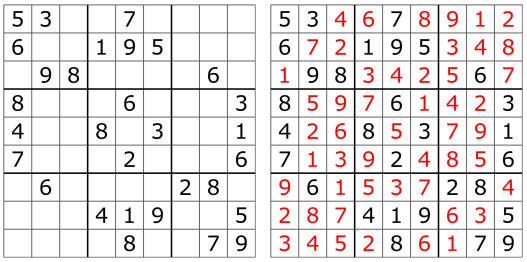

Figura 1: Puzzle sudoku e sua solução

A palavra sudoku é de origem japonesa, sendo a junção de *su*, número, e *doku*, único. Apesar disso, o quebra-cabeça como conhecemos não foi inventado no Japão, mas sim pelo americano Howard Garnes, em 1979 [1].

Nesse trabalho, vamos apresentar um programa que recebe um problema sudoku e retorna a sua solução, caso ela exista.

2 Método

2.1 Busca em espaços de estados

A busca em espaços de estados é um processo utilizado em ciência da computação para resolver problemas de forma sistemática, explorando todas as possíveis soluções até encontrar uma solução válida.

Um "estado"em busca em espaços de estados é uma representação específica do problema em questão. Em problemas como o Sudoku, um estado é uma configuração particular do tabuleiro, incluindo as células preenchidas e as células vazias. A transição entre um estado e outro ocorre quando um número é inserido em uma célula vazia, alterando assim a configuração do tabuleiro.

Nesse sentido, dado um estado inicial, utilizamos a abordagem de busca em espaços de estado para explorar as possíveis configurações deriváveis até encontrarmos uma solução. A busca prossegue até que todos os estados válidos sejam explorados, permitindo-nos determinar se o sudoku é solucionável ou não.

2.2 Algoritmos de busca

Para este trabalho, foram implementados diferentes algoritmos de busca. Em particular, algoritmos de busca podem ser classificados em duas categorias: algoritmos de busca sem informação e algoritmos de busca com informação.

No primeiro caso, os algoritmos de busca sem informação exploram o espaço de estados de forma cega, sem usar informações específicas sobre o problema para orientar a busca. Eles geralmente

consideram todas as possíveis opções de maneira igual, sem priorizar caminhos que possam levar mais rapidamente à solução.

Por outro lado, os algoritmos de busca com informação usam informações específicas sobre o problema para orientar a busca, priorizando caminhos que parecem mais promissores em direção à solução. Eles usam uma heurística para avaliar a qualidade de cada ação, estimando o custo de alcançar o objetivo a partir de um determinado estado.

2.3 Grafos e árvores

Para realizar a busca em espaços de estados, utilizamos algoritmos de busca em grafos. Um grafo, no contexto da busca em espaços de estados, é uma representação visual das transições entre os diferentes estados do problema. Cada vértice do grafo armazena um estado possível do problema, enquanto as arestas indicam as transições possíveis entre esses estados. Em outras palavras, os vértices contém informações sobre os estados do problema, e as arestas representam as ações ou movimentos que levam de um estado para outro.

Em especial, se um grafo é conexo e acíclico, então ele é chamado de árvore. A solução do sudoku não gera ciclos no grafo porque cada célula do tabuleiro é preenchida apenas uma vez e, portanto, não há necessidade de revisitar estados anteriores durante a busca. Além disso, o grafo é conexo, pois cada estado é acessível a partir de qualquer outro estado por meio das transições permitidas. Assim, os estados e transições gerados durante a busca da solução para o problema do sudoku gera uma árvore.

A figura 2 ilustra uma árvore. Essa representação visual facilita a compreensão das relações entre os diferentes estados e as possíveis transições entre eles.

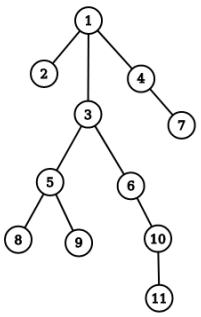

Figura 2: Representação gráfica de uma árvore

2.4 Implementação

O programa desenvolvido para solucionar o problema proposto consiste em um tipo abstrato de dados (TAD) `Solver`, que implementa os algoritmos de busca, além de métodos auxiliares. A tabela 1 apresenta os algoritmos de busca implementados para a solução do problema proposto. Além disso, o programa conta com o *namespace* `grid`, que implementa todas as funções utilizadas sobre a matriz sudoku, como a verificação de se um determinado número é válido em uma posição específica e se uma determinada matriz representa a solução do problema.

A representação dos estados e transições é feita através da estrutura de dados `Graph`, implementada pelo autor. O grafo é construído à medida em que os algoritmos de busca o atravessam. Além disso, estados já explorados (vértices) são apagados logo após gerarem novos estados (vértices filhos), uma vez que não precisamos analisá-los novamente. Essa abordagem se mostrou bastante eficiente em

Tabela 1: Algoritmos de busca implementados

Busca sem informação	Busca com informação
Breadth-First Search (BFS)	A* Search
Iterative Deepening Depth-First Search (IDDFS)	Greedy Best-First Search (GBFS)
Uniform-Cost Search (UCS)	

termos de memória para Sudoku difíceis, isto é, aqueles que exigem a expansão de um grande número de estados até que a solução seja encontrada.

Ademais, todas as estruturas de dados empregadas na solução do problema (fila, fila de prioridade mínima, pilha, map etc) foram implementadas pelo autor.

2.4.1 Breadth-First Search

A BFS começa a busca a partir de um vértice raiz e expande a exploração gradualmente, seguindo uma abordagem em largura, ou seja, visitando todos os vizinhos diretos do vértice atual antes de avançar para os vizinhos dos vizinhos. Isso a torna ideal para encontrar o caminho mais curto entre dois vértices em um grafo, no entanto exige um alto custo de memória para grafos com um grande número de vértices e arestas, uma vez que todos os vértices de um mesmo nível são armazenados em memória enquanto não são explorados.

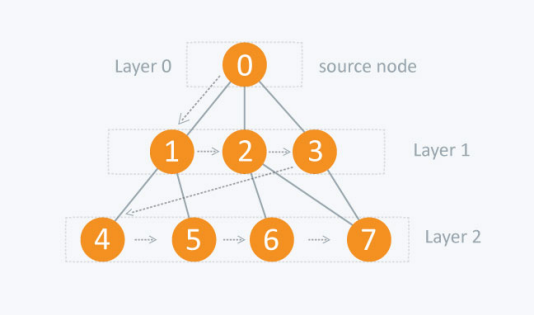

Figura 3: Por construção, a BFS percorre o grafo em camadas, visitando todos os vizinhos de cada vértice antes de iniciar uma nova camada

A implementação da BFS realizada neste trabalho utiliza uma estrutura de dados conhecida como fila, fundamental para armazenar os vértices de um determinado nível enquanto eles não são explorados.

2.4.2 Iterative Deepening Depth-First Search

O IDDFS é uma técnica de busca em grafos que combina as vantagens do algoritmo de busca em profundidade (DFS) e de busca em largura (BFS). Ele é útil para encontrar soluções em espaços de busca muito grandes ou infinitos, como é o caso do Sudoku.

O IDDFS é uma versão iterativa do DFS, onde a profundidade máxima de busca é gradualmente aumentada em cada iteração. Ele começa com uma profundidade máxima de 0 e gradualmente aumenta até encontrar o objetivo ou até esgotar todas as possibilidades. Isso é feito de forma incremental, ou seja, o algoritmo começa uma busca em profundidade limitada a uma profundidade máxima, e se não encontrar a solução, aumenta o limite de profundidade[1] e realiza outra busca.

[1]Para o problema do sudoku em particular, a profundidade máxima é 81, visto que cada estado altera apenas

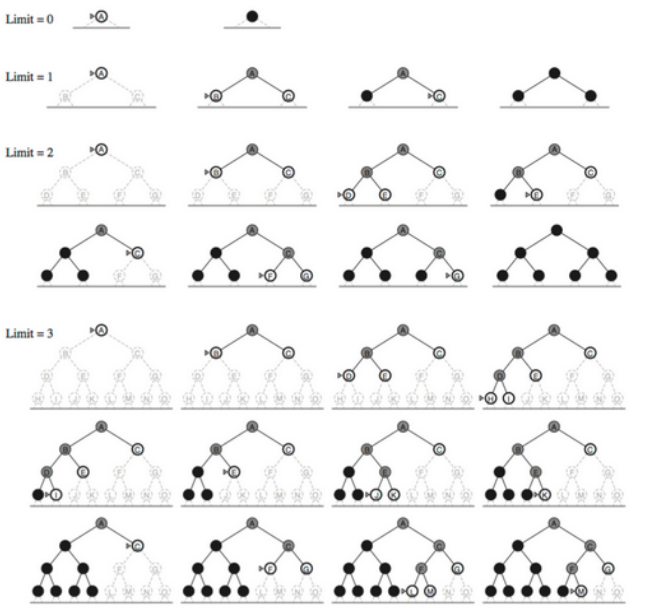

Figura 4: O algoritmo funciona iterativamente, definindo um limite de busca e aplicando a técnica da busca em profundidade

Por exemplo, em um problema de busca em um grafo com um espaço de estados muito grande, o IDDFS pode ser mais eficiente do que a busca em largura porque evita o alto consumo de memória associado com a busca em largura, enquanto mantém a garantia de encontrar a solução mais rasa possível, como faz a BFS.

A implementação do IDDFS geralmente é feita utilizando uma abordagem recursiva, que não exige estrutura de dados auxiliar para armazenar os vértices enquanto a exploração do grafo é realizada. No entanto, para os fins do presente programa, o autor considerou mais fácil implementar uma versão não recursiva, utilizando a estrutura de dados pilha para armazenar os vértices durante a busca.

2.4.3 Uniform-Cost Search

A busca de custo uniforme é uma variação do famoso algoritmo de Dijkstra, utilizado para encontrar o menor caminho de um vértice de origem até todos os outros vértices do grafo. A UCS, por outro lado, concentra-se em encontrar o menor caminho entre um vértice de origem e um vértice de destino, parando a busca logo que o objetivo é atingido. Além disso, a principal vantagem da UCS em relação ao algoritmo de Dijkstra é o consumo de memória. Enquanto o algoritmo de Dijkstra armazena todos os vértices em memória no início de sua execução, a UCS armazena somente aqueles vértices que explorará em um futuro próximo, consumindo muito menos memória em cenários onde o grafo pode ser formado por um grande número de vértices.

Para realizar tal feito, a UCS, assim como o algoritmo de Dijkstra, explora primeiro os vértices que têm o menor custo até então. Para tanto, é necessário utilizar uma estrutura de dados conhecida como fila de prioridade mínima, que é uma fila que retorna os objetos de acordo com alguma função de prioridade, que no nosso caso é o custo.

uma célula do tabuleiro e temos 9x9 células

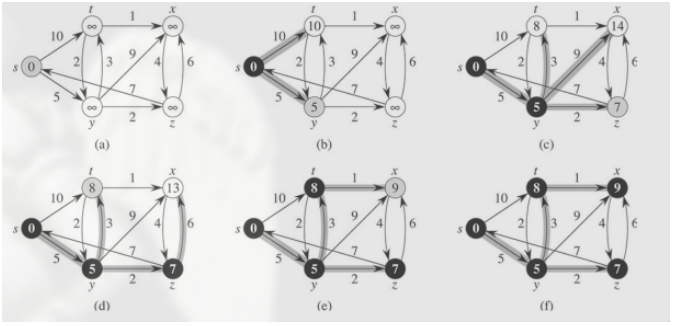

Figura 5: Execução do algoritmo de Dijkstra em um grafo ponderado. Se utilizarmos a UCS para encontrar o menor caminho entre o vértice 0 e o 7, executaríamos até o passo (d)

2.4.4 A* Search

O Algoritmo A* é uma técnica de busca informada amplamente utilizada em problemas de otimização. Ele combina a eficiência da busca de custo uniforme com a capacidade de guiar a busca em direção ao objetivo usando uma heurística, a qual estima o custo restante para alcançar o objetivo a partir de cada estado.

Enquanto a busca de custo uniforme considera como função de custo $g(n)$ somente o valor do caminho do vértice inicial até o vértice atual, o A* considera ainda o valor da heurística dada pela função $h(n)$, e sua função de custo é dada por $f(n) = g(n) + h(n)$. Assim, o algoritmo prioriza os vértices que têm maior probabilidade de levar a uma solução ótima, o que resulta em uma busca mais eficiente.

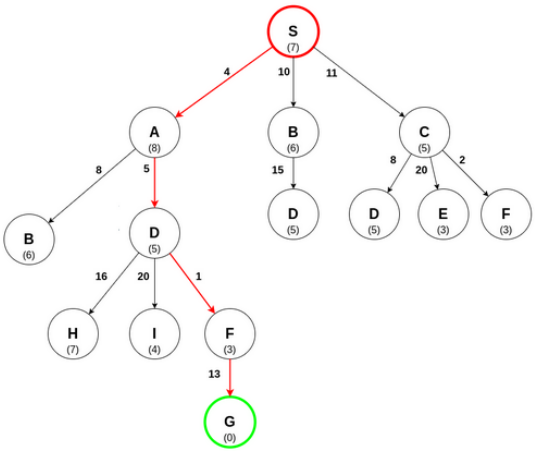

Figura 6: Exemplo de execução do A*. O valor entre parênteses dentro dos círculos foi dado pela função heurística.

Assim como a UCS, o algoritmo A* toma-se de uma fila de prioridade mínima para priorizar a exploração dos vértices que apresentam menor custo.

2.4.5 Greedy Best-First Search

O GBFS é uma variação do algoritmo BFS, mas, ao invés de expandir todos os vértices no mesmo nível antes de prosseguir para o próximo nível, ele escolhe o vértice mais promissor de acordo com uma heurística específica. A função de custo deste algoritmo não leva em consideração o custo de

atravessar uma aresta, guiando sua busca somente pelo valor dado pela heurística. Portanto, sua função de custo é $f(n) = h(n)$.

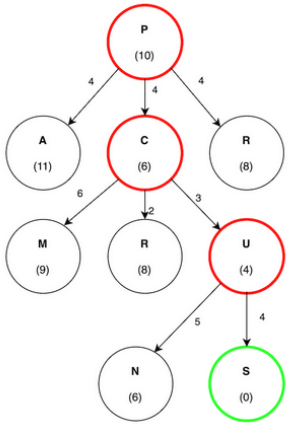

Figura 7: Exemplo de execução do GBFS. O valor entre parênteses dentro dos círculos foi dado pela função heurística.

O algoritmo também utiliza-se de uma fila de prioridade mínima para priorizar a exploração dos vértices que apresentam menor custo.

2.5 Heurísticas

Como dito anteriormente, os algoritmos de busca com informação empregam heurísticas para avaliar a qualidade de cada ação em direção ao objetivo. Nesse sentido, para cada um dos dois algoritmos de busca com informação implementados, empregamos duas diferentes heurísticas.

2.5.1 Heurística para o A* Search

A heurística desenvolvida para o algoritmo A* Search baseia-se na quantidade de valores possíveis para uma determinada célula da matriz. Dessa forma, após criar um novo vértice por meio do processo de expansão, determinamos quantos valores possíveis são válidos para a célula que este vértice modificará. A motivação para essa estratégia foi a observação de que:

1. Para cada valor possível, um novo estado será criado. Assim, haverá mais ramificações na árvore de busca e um maior custo computacional para encontrar a solução;
2. A medida que a quantidade de células vazias diminui, menor será a quantidade de valores disponíveis para um determinado vértice, visto que a chance de haver valores inválidos é maior.

A observação (1) é interpretada como um indício de que estamos distantes da solução almejada. Por outro lado, a observação (2) pode indicar que estamos mais próximos da solução. Para visualizar isso, imagine um grid 9x9 com apenas uma célula vazia. Considerando que o estado atual do grid é válido, isto é, não há nenhuma célula que infringe alguma das regras do Sudoku, se aquele estado puder ser solucionado, haverá exatamente um único número possível. Assim, a observação (2) nos aproxima ou da solução ou de um estado parcialmente solucionado mas que não pode gerar um estado solução.

Logo, atribuímos um valor de heurística menor quando há menos possibilidades de valores para uma determinada célula, com o objetivo de determinar rapidamente se aquele caminho leva ou não a uma solução

2.5.2 Heurística para o Greedy Best-First Search

A heurística escolhida para o algoritmo GBFS considera a quantidade de células ainda vazias na matriz, atribuindo menor custo aos estados que apresentam menos células vazias. A escolha dessa heurística baseia-se na observação imediata de que um grid com 5 células vazias está mais próximo de uma possível solução do que um grid com 10 células vazias.

2.6 Custo de travessia de arestas

Algoritmos de busca geralmente consideram o custo de atravessar uma determinada aresta durante sua execução. No contexto da busca em espaços de estado, atravessar uma aresta é equivalente ao processo de expandir um vértice e ir até seu filhos.

Alguns algoritmos, como a BFS, não consideram o custo de atravessar uma aresta durante a busca. Por outro lado, algoritmos como a UCS consideram o custo de atravessar uma aresta. Caso o custo seja igual para todas as arestas do grafo, então esses algoritmos podem degenerar-se em outros, como é o caso da UCS que, quando todas as arestas têm o mesmo custo, se degenera em uma BFS.

Portanto, para esses casos, definimos uma função que atribui um custo pseudo-aleatório inteiro no intervalo $[1, 10]$ para cada aresta do grafo.

3 Análise de complexidade

O problema geral de resolver quebra-cabeças sudoku em tabuleiros $n^2 \times n^2$ com sub-regiões $n \times n$ é sabidamente NP-Completo [2, 3], ou seja, é um problema muito difícil de resolver e que não se sabe de nenhum algoritmo que encontre a solução em tempo polinomial, mas que, dada uma solução, podemos verificar sua validade rapidamente [4]. A estratégia, então, é explorar todas as soluções possíveis até encontrar uma que seja válida, que é exatamente o que os algoritmos de busca em espaços de estado implementados neste trabalho fazem.

Os algoritmos BFS e IDDFS possuem complexidade de tempo e espaço $O(b^d)$ [5], onde b é a profundidade da solução mais rasa, isto é, a distância[2] entre o estado inicial (vértice raiz da árvore) até o vértice que contém a solução, e d é o fator de ramificação (*branching factor*), a medida que descreve o número médio de filhos que um vértice possui em um grafo. Já o algoritmo UCS possui complexidade de tempo e espaço $O(b^{1+\lfloor \frac{C*}{\epsilon} \rfloor})$ [5], onde b é a profundidade da solução mais rasa, $C*$ é o custo da solução ótima (solução de menor custo) e ϵ é o custo mínimo de ir de um estado à outro na árvore.

Por outro lado, a complexidade dos algoritmos de busca informada depende da qualidade da heurística escolhida [5]. Para o caso de a heurística ser consistente, isto é, ser capaz de sempre encontrar o melhor caminho, então o algoritmo A* tem complexidade de tempo e espaço no pior caso $O(b^d)$ [6] e o algoritmo GBFS no pior caso tem complexidade de tempo e espaço $O(|V|)$, onde V é a quantidade de vértices no grafo [5].

Assim, a tabela 2 apresenta a complexidade de tempo e espaço de cada um deles.

[2] A Distância entre um vértice A e um vértice B é a quantidade de vértices no caminho que leva de A até B.

Tabela 2: Algoritmos de busca e suas complexidades no pior caso

Algoritmo	Complexidade de tempo	Complexidade de espaço
BFS	$O(b^d)$	$O(b^d)$
IDDFS	$O(b^d)$	$O(b^d)$
UCS	$O(b^{1+\lfloor \frac{C*}{\epsilon} \rfloor})$	$O(b^{1+\lfloor \frac{C*}{\epsilon} \rfloor})$
A* Search	$O(b^d)$	$O(b^d)$
GBFS	$O(\mid V \mid)$	$O(\mid V \mid)$

4 Análise Experimental

Para realizarmos a análise comparativa dos algoritmos de busca implementados, utilizamos um script[3] python responsável por gerar quebra-cabeças sudokus 9x9, de forma que o problema gerado tenha apenas uma solução. Para que a análise comparativa faça sentido, os problemas necessariamente devem ter apenas uma solução, uma vez que os algoritmos exploram o grafo de forma diferente. Para o caso de os problemas apresentarem mais de uma solução, corremos o risco de comparar soluções encontradas em profundidades diferentes da árvore. Assim, como a complexidade dos algoritmos leva em consideração a profundidade da solução, poderíamos erroneamente comparar cargas de trabalho diferentes como se fossem iguais.

Uma vez gerado os casos de testes, decidimos classificá-los em níveis de dificuldade. Em geral, a BFS é o algoritmo dentre os algoritmos implementados neste trabalho que expande uma maior quantidade de estados até achar a solução. Assim, utilizamos a BFS para classificar cada problema gerado em termos de quantos estados foram expandidos, conforme a tabela 3.

Tabela 3: Classificação de dificuldade de acordo com o total de estados expandidos pela BFS

Nível	Estados expandidos	Tamanho da amostra
Super fácil	[0, 1000)	5
Fácil	[1000, 25000)	117
Médio	[25000, 100000)	43
Difícil	[100000, $+\infty$)	35

O script utilizado para gerar os casos de teste emprega uma estratégia onde não é possível determinar a "dificuldade"do problema gerado. Assim, durante o processo de classificação de acordo com os critérios estabelecidos, o tamanho da amostra para cada nível ficou diferente.

Para a análise comparativa, decidimos avaliar o tempo até encontrar a solução e a quantidade de estados expandidos por cada algoritmo. Para tanto, executamos os mesmo casos de testes classificados para cada um dos algoritmos, capturando o tempo de execução e a quantidade de estados expandidos e fazendo a média simples.

4.1 Análise comparativa pelo total de estados expandidos

A figura 8 apresenta o comparativo de número médio de estados expandidos por cada algoritmo em cada nível de dificuldade. Podemos observar que para os casos super fácil e fácil, o algoritmo A* é o que obteve o melhor desempenho, batendo até mesmo o GreedyBFS (GBFS), que também é de busca informada.

Por outro lado, para os casos de nível médio e difícil, os algoritmos de busca informada não apresentaram um desempenho melhor que os algoritmos de busca não informada. Isso pode significar que

[3]O script foi adaptado de www.101computing.net/sudoku-generator-algorithm/

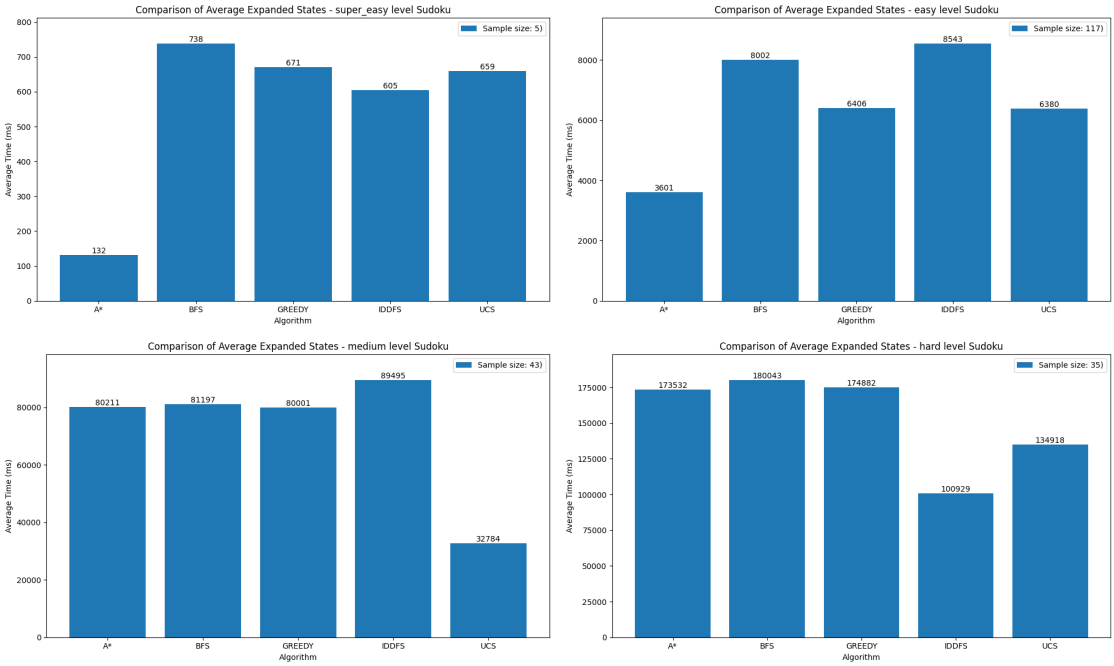

Figura 8: Média de estados expandidos

nossas heurísticas operam de forma razoável em casos mais fáceis, mas têm dificuldades em direcionar os algoritmos de busca informada para o melhor caminho quando a solução para o problema é mais difícil.

No geral, o UCS foi o algoritmo que apresentou os melhores resultados de forma consistente, igualando o desempenho com o algoritmo GBFS nos casos fáceis e batendo os algoritmos de busca informada nos casos difíceis.

4.2 Análise comparativa pelo tempo de execução

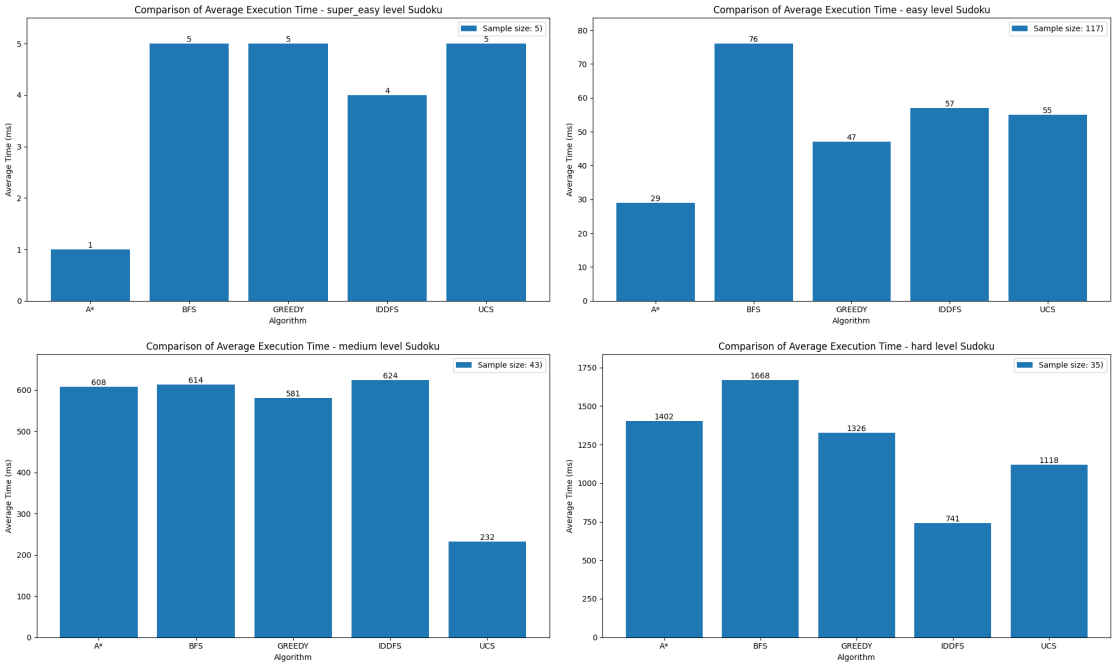

Figura 9: Média de tempos de execução

A figura 9 apresenta os tempos médios de execução de cada algoritmo em cada nível de dificuldade. É evidente a semelhança entre os gráficos da figura 9 com os já apresentados na figura 8. Isso demonstra que o tempo de execução está relacionado diretamente com o total de estados expandidos, uma vez que a função sucessora é a mesma para todos os algoritmos e, portanto, apresenta o mesmo custo computacional para ser executada.

5 Conclusão

Nesse trabalho, apresentamos o problema do Sudoku e um programa que recebe e soluciona tal problema, caso uma solução exista. Além disso, abordamos a busca em espaços de estado e os algoritmos que utilizam este método para encontrar uma solução para o referido problema.

O desenvolvimento deste trabalho possibilitou ao aluno o aprendizado sobre os conceitos teóricos envolvidos na solução do problema, além do desenvolvimento de habilidades práticas relacionadas à implementação dos algoritmos empregados.

Referências

[1] Gustavo Santos-García e Miguel Palomino. "Solving Sudoku Puzzles with Rewriting Rules". Em: *Electronic Notes in Theoretical Computer Science* 176.4 (2007). Proceedings of the 6th International Workshop on Rewriting Logic and its Applications (WRLA 2006), pp. 79–93. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2007.06.009. URL: https://www.sciencedirect.com/science/article/pii/S1571066107005142.

[2] Takayuki YATO e Takahiro SETA. "Complexity and Completeness of Finding Another Solution and Its Application to Puzzles". Em: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E86-A (2003).

[3] Eline Sophie Hoexum. "Revisiting the proof of the complexity of the sudoku puzzle". Em: (2020).

[4] Thomas H. Cormen et al. *Algoritmos - Teoria e Prática*. Vol. 1. Elsevier Brasil, 2017. ISBN: 978-85-3527-179-9.

[5] Stuart J. Russell e Peter Norvig. *Inteligência Artificial - Uma Abordagem Moderna*. Vol. 1. GEN LTC, 2022. ISBN: 978-85-9515-887-0.

[6] Wikipedia contributors. *A* search algorithm*. 2024. URL: https://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=1221291584 (acesso em 02/05/2024).