# A Comparative Analysis of Search Algorithms for Sudoku Solving

Guilherme S. C. Caporali (2021031955)
*guilhermesoeiro@dcc.ufmg.br*
*DCC, UFMG*

**Abstract**

This project explores the efficacy of five search algorithms—Breadth-First Search (BFS), Iterative Deepening Search (IDS), Uniform Cost Search (UCS), Greedy Best-First Search (GBFS), and A*—in solving Sudoku puzzles. Each algorithm is implemented and tested on a variety of Sudoku instances to evaluate their performance in terms of computational efficiency. Results indicate distinct trade-offs among the algorithms, with some prioritizing memory usage while others focus on speed. By comparing their performance metrics, this study offers insights into the strengths and weaknesses of each algorithm, judging if it is a good choice given the problem's properties.

# 1 Sudoku

## 1.1 Rules

Sudoku is a single-player logic-based puzzle game played on a 9x9 grid divided into nine 3x3 subgrids. The objective is to fill the grid with digits from 1 to 9, ensuring that each row, each column, and each 3x3 subgrid contains all the digits from 1 to 9 without repetition. At the beginning of the game, a partially filled grid is provided, with some cells already containing numbers. The player's task is to use logic and deduction to fill in the remaining empty cells, following the constraints of the game. The game is won when the entire grid is filled correctly according to the rules, with each row, column, and subgrid containing the numbers 1 to 9 exactly once. For further information on the game, see the sudoku's wikipedia page.

## 1.2 Implications

The rules of Sudoku present significant implications for its implementation, particularly concerning memory usage and search space exploration. The considerable size of the 9x9 grid necessitates careful management of memory resources, especially when considering breadth search algorithms like Breath-first Search (DFS), Uniform-cost Search (UCS) and A* Search. The branching factor of the search tree is directly influenced by the number of possible digits that can be placed in each empty cell, which typically ranges from 1 to 9. Additionally, the depth of the search tree is equal to the number of empty cells on the initial board. These factors highlight the importance of employing efficient search algorithms and heuristic strategies to navigate the solution space effectively while mitigating computational overhead.

# 2 Methodology

In order to test the implementation and acquire metrics on it's performance, 3 cases were selected: an easy initial state, a medium initial state and a hard initial state. After running all 5 algorithms on each of the 3 instances of the problem, it was possible to compute metrics like the number of expanded states and the execution time. These statistics were used to explain and explore the behaviour observed.

# 3 Implementation & Results

In our implementation, each game state is represented by a 9x9 grid or board, where each cell contains a digit from 1 to 9 or is empty, denoted by 0. To optimize memory usage and improve performance, we represent the board as a contiguous 81-element vector of **u_int8_t**, leveraging the locality principle to enhance cache

efficiency and reduce memory overhead. Each algorithm may implement a different successor function to generate possible moves from a given game state, exploring the search space in different ways. Additionally, a verification function ensures the validity of each game state by checking for the absence of zeros on the board (there is no need the verify the other rules, if the successor function only generate valid moves), ensuring adherence to Sudoku rules. Notably, heuristics employed by A* and Greedy Best-First Search (GBFS) differ, influencing their search strategies and solution paths. All algorithms are implemented in C++ and compiled using GNU's g++ compiler (the compilation used the **-03** optimization flag in order to minimize inefficiencies caused by poorly written code), leveraging the language's efficiency and flexibility for algorithmic implementations.

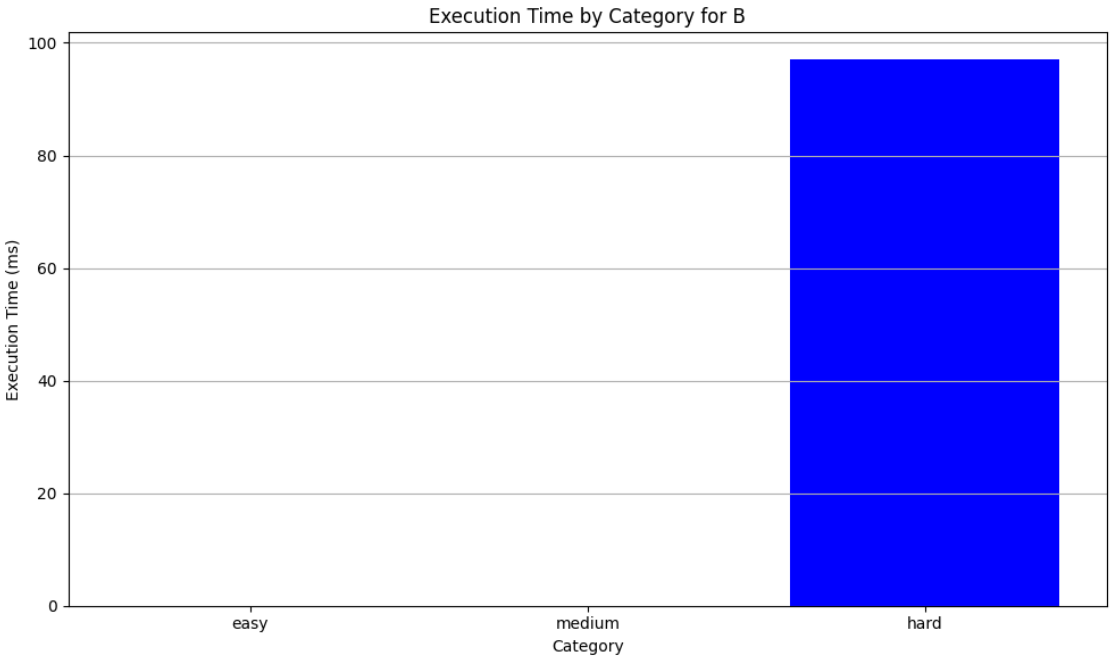
All the code used in this project can be found at the project's github repository.

## 3.1 Breadth-first search (B)

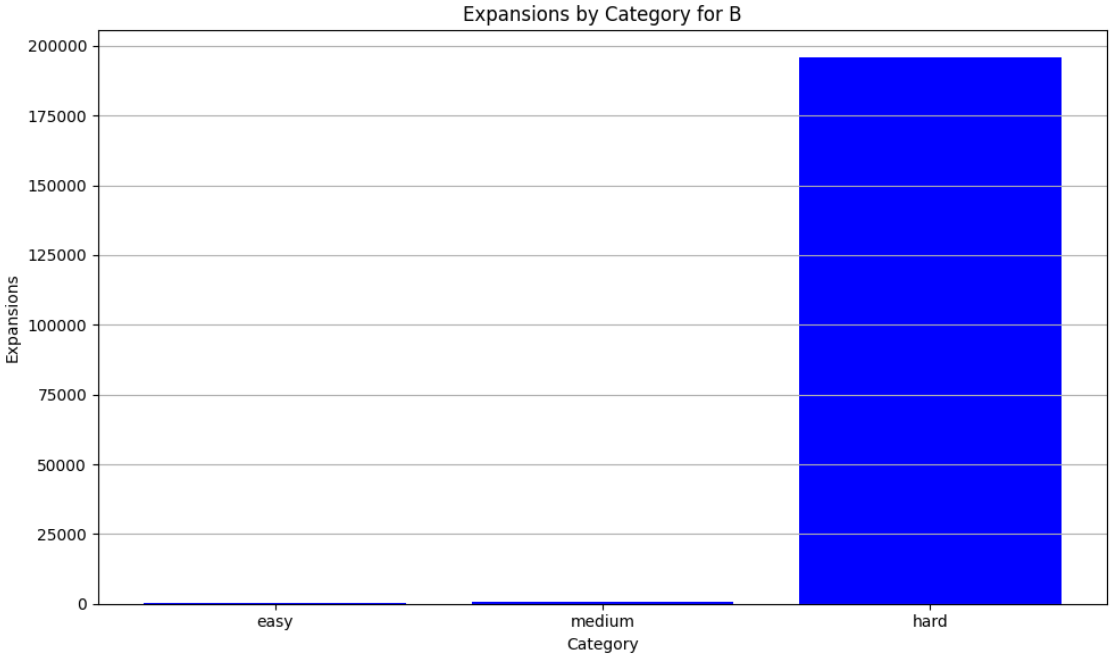
### 3.1.1 Implementation

The implementation of the Breadth-first search used the C++ standard library's queue to represent the frontier set (set of states to be visited). The successor function used generates the valid moves of the next empty cell, instead of generating all valid moves on the state. This optimization is necessary because of the exponential complexity of this algorithm combined with the high branching factor that would occur if all valid moves were generated at each step.

### 3.1.2 Results




The results affirm the exponential behaviour of the algorithm. On the first figure, we can see that the execution time is 0ms for both easy and medium cases, increasing to around 100ms on the hard case. Something similar happens on the number of expanded states, with the easy and medium cases being negligent when compared to the hard case. This is due to the nature of the algorithm, which expands all states in the same level of the tree of possibilities. This property combined with a solution that lies deep in the tree and a high branching factor resulted in a scalability problem that had to be addressed by limiting the branching factor via expanding only the valid moves of the next empty position on the board, instead of all empty positions.



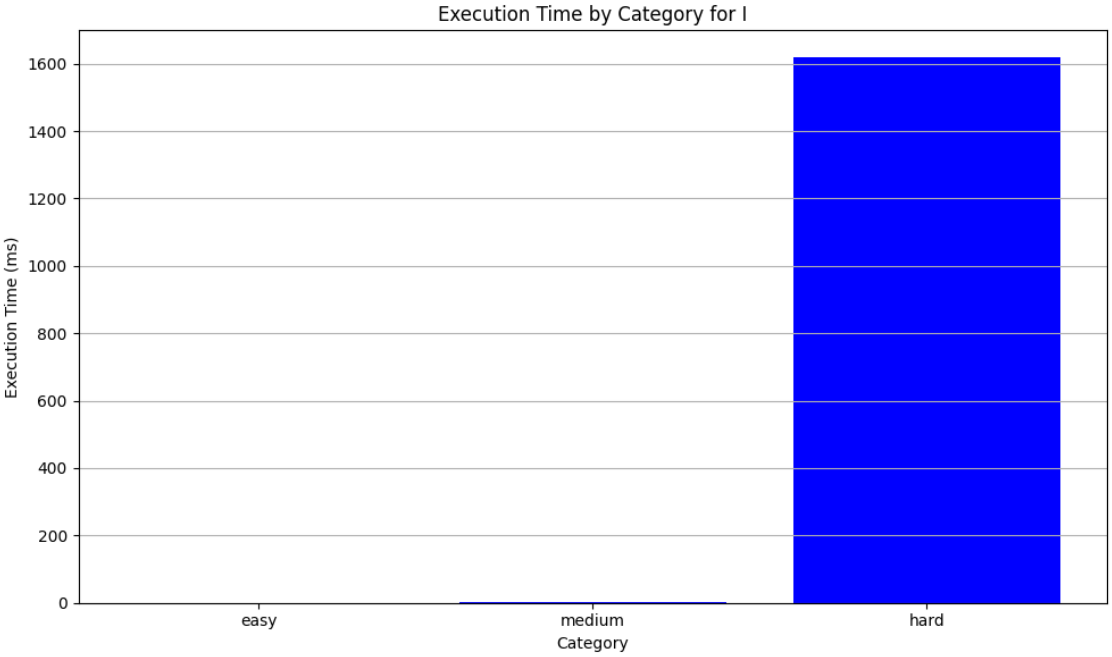

## 3.2 Iterative deepening search (I)

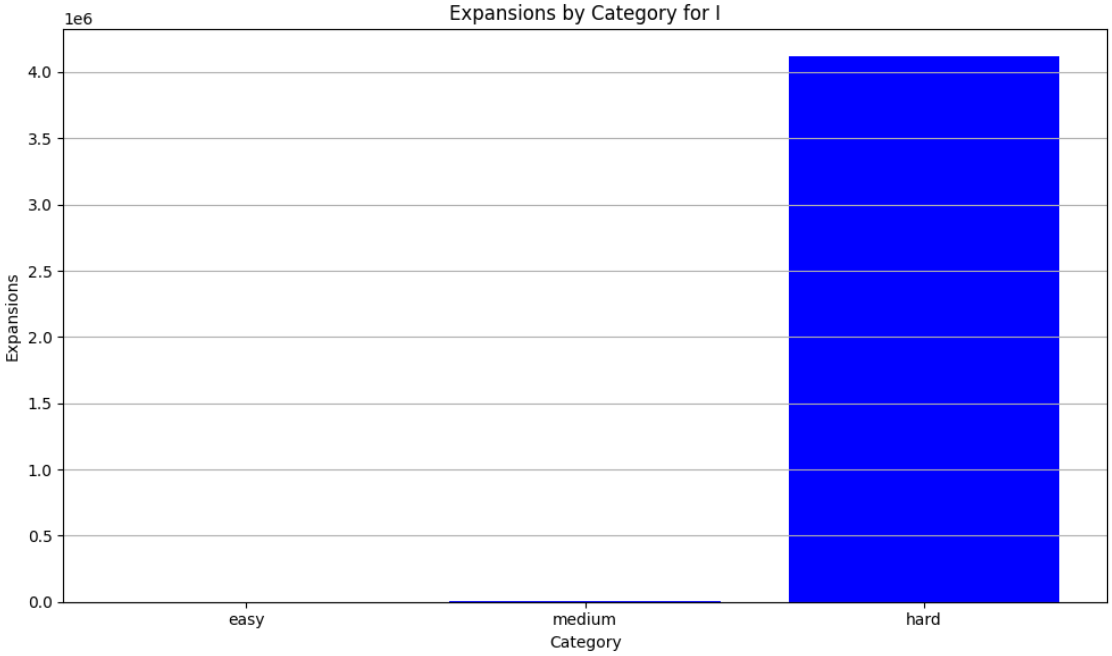
### 3.2.1 Implementation

To implement the Iterative deepening search algorithm, the recursive approach was really useful. This decision was made considering the fact that there is no need to copy the state of the game, since this algorithm uses backtracking. This approach makes the implementation easier and more memory efficient, without compromising speed.

### 3.2.2 Results

The results of the experiment revealed that the performance of this algorithm is worse than the previous one in terms of execution time and number of expansions. This is happens because of the trade-off this method uses, favouring a decrease in memory usage by expanding each node more than once (recalculating the entire path to each state), therefore not having to store it in memory. The worse performance in speed is compensated by the low memory usage, making this algorithm a good choice for some environments.


Execution Time by Category for I
Execution Time (ms)
0
200
400
600
800
1000
1200
1400
1600
easy
medium
hard
Category

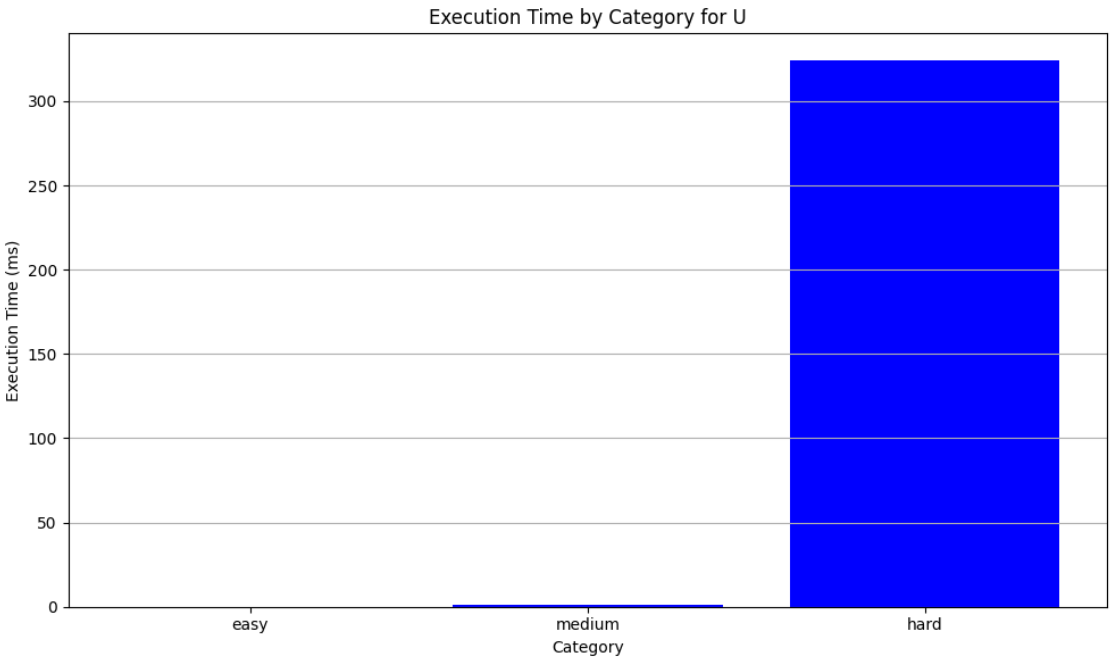

Expansions by Category for I
1e6
Expansions
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
easy
medium
hard
Category

## 3.3 Uniform-cost search (U)

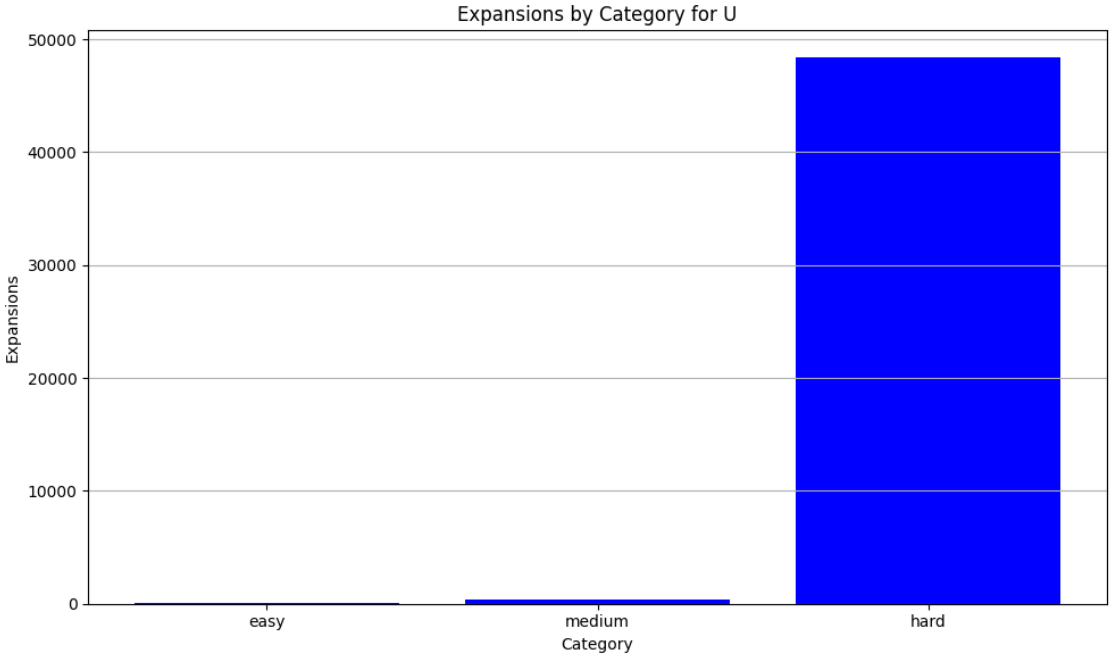
### 3.3.1 Implementation

The implementation of the Uniform-cost search algorithm is similar to the Breadth-first search algorithm implementation, except for the data structure used. For the UCS, the C++ standard library's priority queue was used. The comparison used on the priority queue was designed to prioritize the move that results in the least next possible moves. The idea is to reduce the effective branching factor during the search. It was necessary to use the same optimization as in the BFS algorithm because of the same memory usage issue.

### 3.3.2 Results




As the data shows, using a priority queue instead of a regular queue caused the search to expand less states, reducing the number of steps taken on the search. Despite that, the UCS execution time was worse than the BFS execution time. This happens because of the cost to compute the priority of a node. It is important to observe that a better way of comparing the states could make the UCS better than the BFS in all aspects considered.



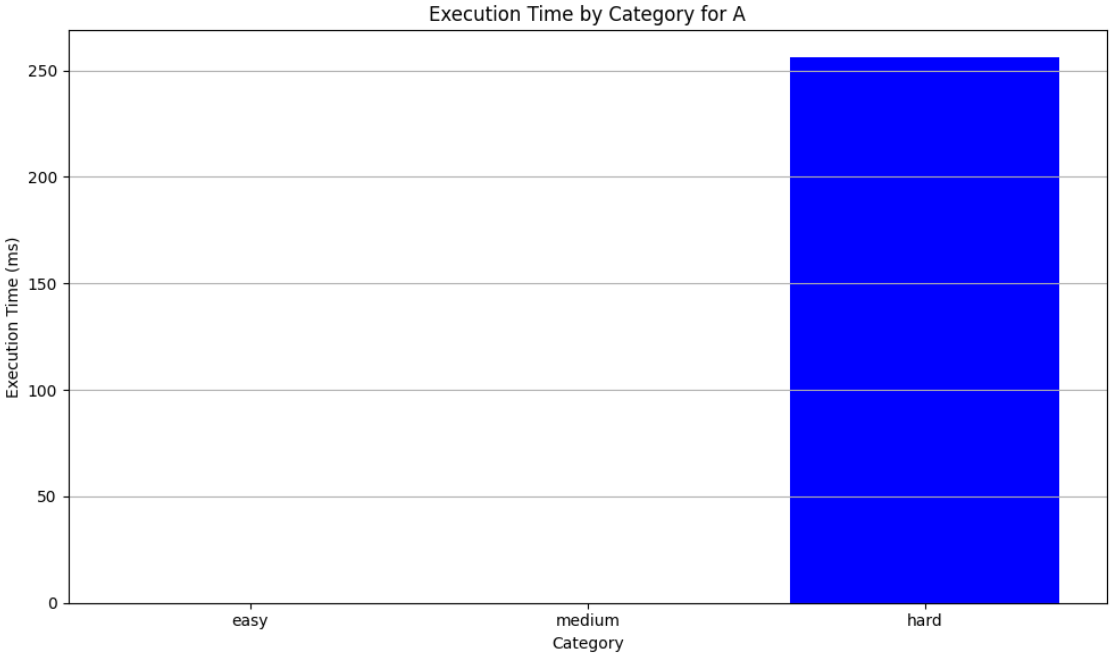

## 3.4 A* search (A)

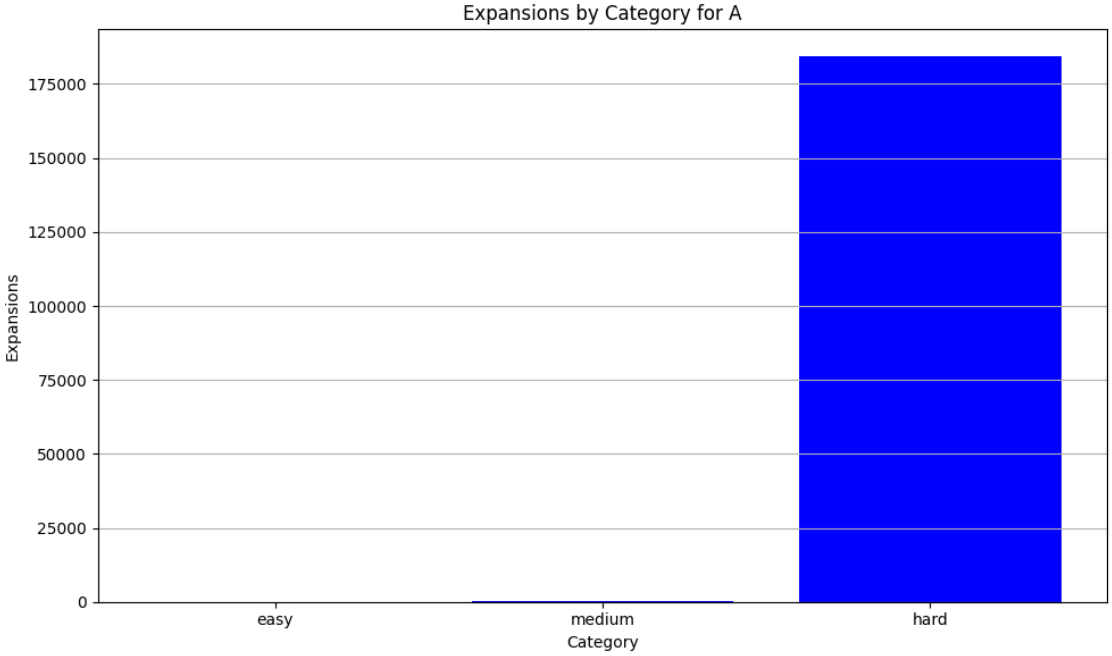
### 3.4.1 Implementation

The implementation of the A* algorithm is similar to the implementation of the Uniform-cost search. The difference lies in the values stored in the priority queue. In addition to the state of the game, the A* algorithm also stores an heuristic that represents the estimated cost to get from the current node to the solution. In this implementation, the heuristic used was the depth of the state. This way, the algorithm prioritizes the node that are most likely closer to the solution. This works because the solution always lies in the deepest level of the tree.

### 3.4.2 Results

The results of the experiments reveal that the heuristic used was capable of reducing the execution time in comparison with the Uniform-cost search, even though the number of expansions increased. This can be explained by the fact that the chosen heuristic is much easier to calculate than the comparison used on the UCS implementation. Despite this marginal improvement, it is possible to say that the choice of the heuristic could have been better. A superior estimate for the cost would have probably yielded a better performance, which could surpass the UCS in terms of number of expansions, thus execution time.


Execution Time by Category for A
Execution Time (ms)
0
50
100
150
200
250
easy
medium
hard
Category

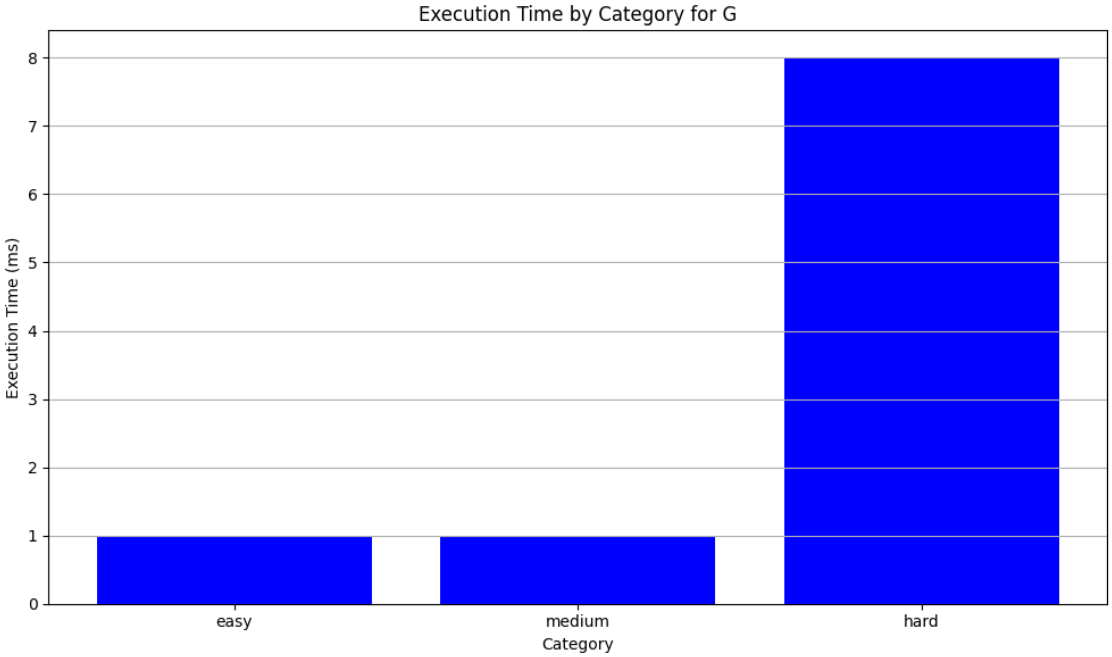

Expansions by Category for A
Expansions
0
25000
50000
75000
100000
125000
150000
175000
easy
medium
hard
Category

## 3.5 Greedy best-first search (G)

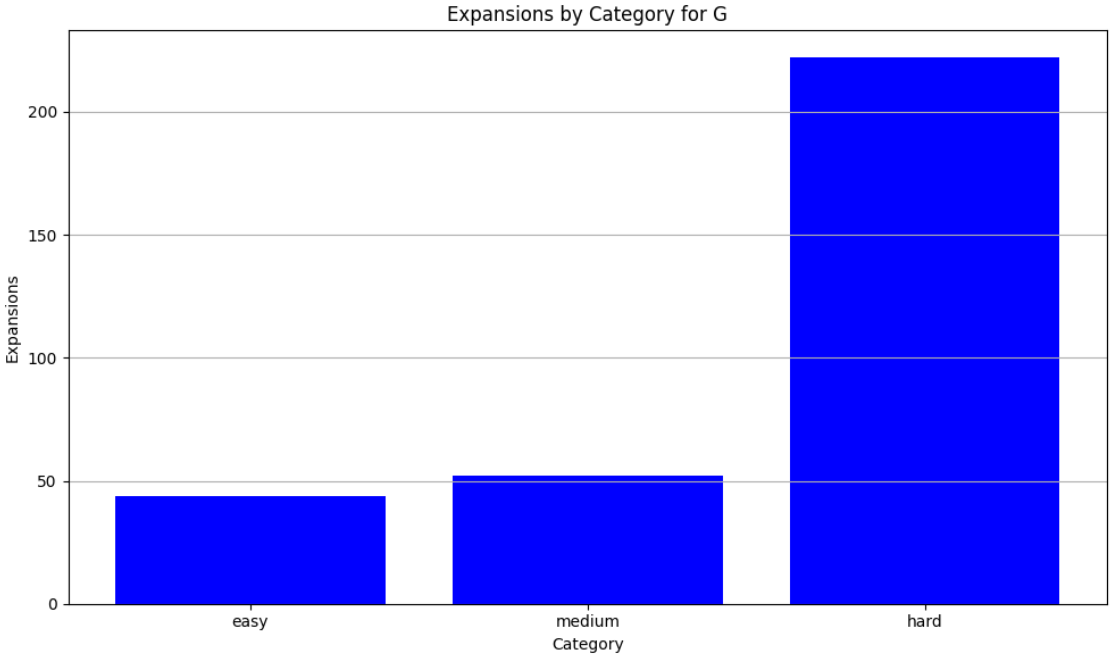
### 3.5.1 Implementation

To implement the Greedy best-first search, a priority queue was used. In this structure were stored the number of possibilities of the last altered cell, giving priority to the smallest of these values. The idea behind this heuristic it to reduce the effective branching factor while increasing the accuracy of the movement. It is also interesting to say that the calculation of this estimate is not expensive, avoiding problems like the one caused by the comparison used in the UCS implementation.

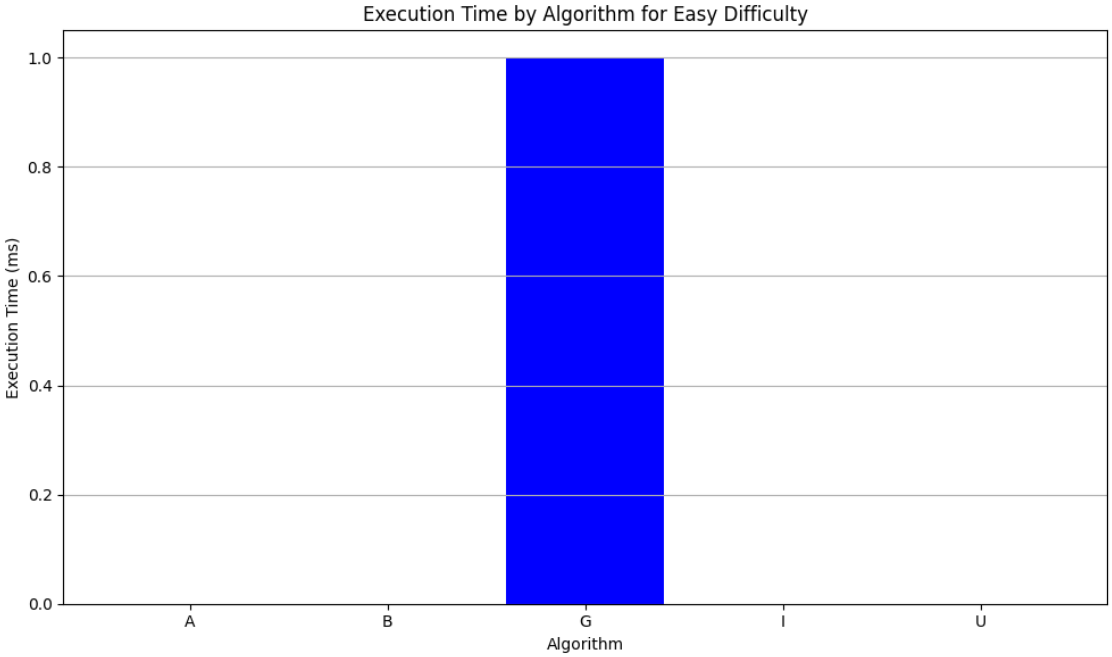
### 3.5.2 Results

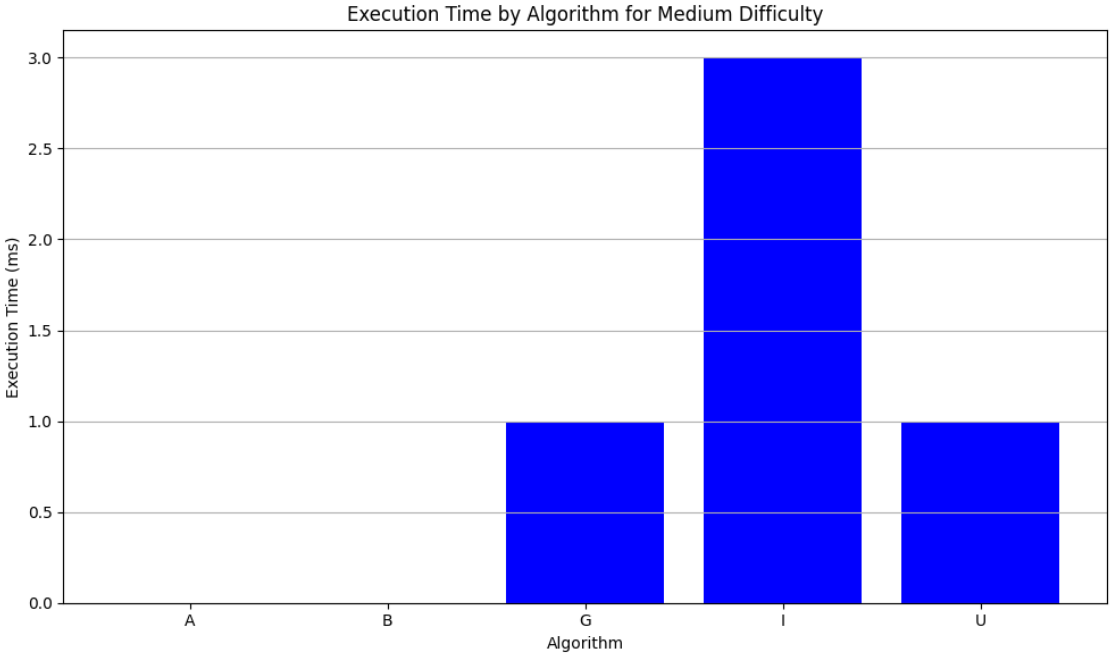


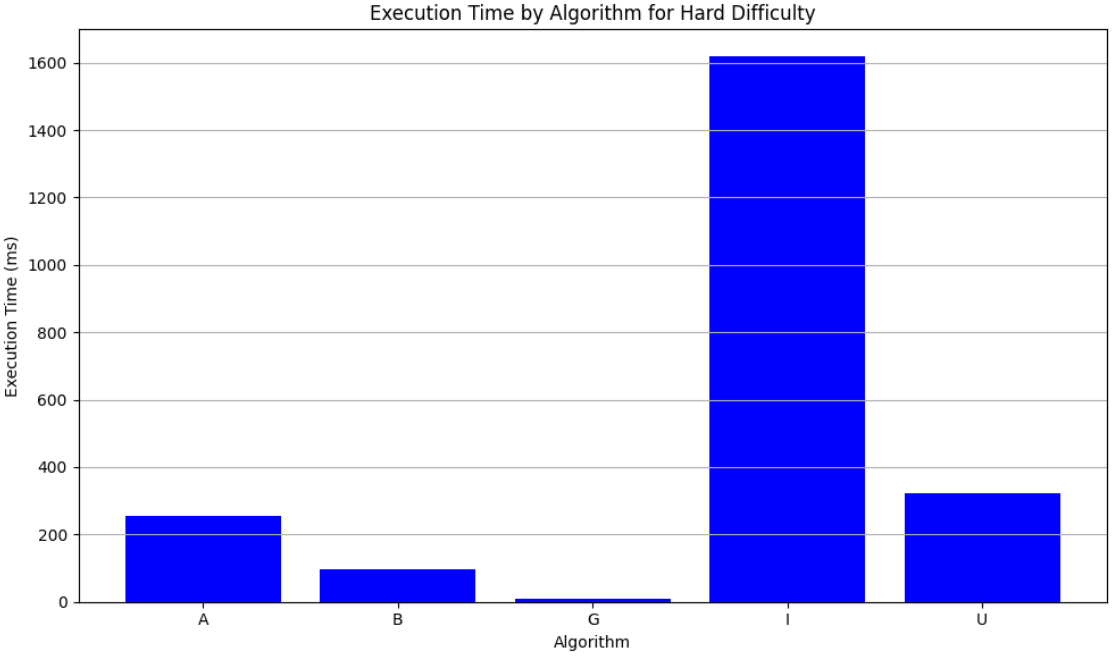

As the plots containing the results of the experiment show, the Greedy Best-first search performed better than all other algorithms presented in this study. A brief analysis of the data informs that the heuristic chosen succeeded on reducing the search space and improve the movement accuracy.




## 3.6 General Comparison




Execution Time by Algorithm for Medium Difficulty
Execution Time (ms)
3.0
2.5
2.0
1.5
1.0
0.5
0.0
A
B
G
I
U
Algorithm


Execution Time by Algorithm for Hard Difficulty
Execution Time (ms)
1600
1400
1200
1000
800
600
400
200
0
A
B
G
I
U
Algorithm

# 4 Conclusion

After analyzing all the data collected and understanding the strengths and weaknesses of the implementation of each algorithm, we can conclude that the Greedy Best-first Search was the fittest method to solving the sudoku game. We can attribute this success to the fact that it explores properties of the game, like the fact that the solution always lies on the same depth and the most restrict movements are more accurate (statistically). To the fail of the other algorithms, we can attribute the misfit with the characteristics of the game, for example, the Breadth-first Search exploring all possibilities and having to store them in memory does not work on a problem with high branching factor, because of high use of memory. On the Uniform-cost Search, the fact that there is no straight-forward way to compare two states makes the method inappropriate, since the comparison of the states is supposed to indicate the best path to the solution. The Iterative Deepening Search cannot be classified as a misfit for this problem. It's disadvantage turns out to be a trade-off that can be really useful when working on low memory environments, replacing memory use with extra computation. And for the A* Search, the cause of the misfit was the poor heuristic choice. A finer choice for the estimate could have resulted on a better performance, with less expansions, thus reduced execution time.