


# Algoritmos de busca aplicados ao Sudoku (com grade simples)


Sudoku, é um jogo baseado na colocação lógica de números. O objetivo do jogo é a colocação de números em cada uma das células vazias numa grade quadrada. O quebra-cabeça contém algumas pistas iniciais, que são números inseridos em algumas células, de maneira a permitir uma indução ou dedução dos números em células que estejam vazias. Cada coluna, linha e região só pode ter um número de cada, sem poder conter números repetidos.

Exemplos de Sudoku 4x4 6x6 9x9

No description has been provided for this image

Utilizamos grafos para modelar o problema, sendo que cada nodo representa um estado do sudoku, aonde o nodo inicial representa o quadro inicial com algumas células já preenchidas, e cada nodo vizinho representando um novo quadro com uma nova célula preenchida com um valor válido, quando não existe um vizinho com um valor válido para aquele nodo, e feito um backtrack, criando um novo nodo com um valor diferente para aquela célula, continuando com o processo até o estado final, aonde todas as células estejam preenchidas com valores válidos

Cada estado é representado por um nodo no grafo

No description has been provided for this image

No problema usamos uma heurística simples, que conta a quantidade de células ainda não preenchidas.

A função de custo entre os estados simplesmente conta a diferença absoluta entre a quantidade de casas em cada estado.

Vamos ver a aplicação das funções ditas acima, além de algumas outras auxiliares que são necessárias para a aplicação dos algoritmos. Primeiro, vamos estabelecer um sudoku de exemplo:

```
In [1]: # === ignore this step, just imports ===
# foi preciso importar de alguma maneira o código do repositório aqui e esse foi
# o jeito mais fácil
import sys
import os

sys.path.append(os.path.join(os.getcwd(), '..'))

from sudoku import *
from algorithms import *

import os
from IPython.core.getipython import get_ipython
# Get the current notebook's path
notebook_path = get_ipython().getoutput('pwd')[0]
# ===

sudoku = [
    [-1, -1, -1, 2],
    [-1, 2, -1, -1],
    [2, 4, 1, 3],
    [3, -1, 2, -1]
]
```

Repare que na estrutura de dados que criamos, as posições da matriz com `-1` representam as casas vazias.

Podemos checar se esse sudoku está finalizado com a função `goal_check`:

```
In [2]: goal_check(sudoku)
```

```
Out[2]: False
```

Veja também a resposta da função para um sudoku válido:

```
In [3]: matrix = [
    [4, 2, 3, 1],
    [1, 3, 4, 2],
    [2, 4, 1, 3],
    [3, 1, 2, 4]
]
goal_check(matrix)
```

```
Out[3]: True
```

A função de heurística nos dará uma estimativa do quão longe estamos da solução final. No caso, como são 8 casas com -1

```
In [4]: heuristic(sudoku)
```

Out[4]: 8

A função de custo, dará a diferença entre dois sudokus diferentes. Veja abaixo que foi definido um novo sudoku bem similar ao anterior, só que com uma casa a mais definida. Logo, o custo entre eles é 1.

```
In [5]: sudoku2 = [
    [4, -1, -1, 2],
    [-1, 2, -1, -1],
    [2, 4, 1, 3],
    [3, -1, 2, -1]
]

cost_between(sudoku, sudoku2)
```

Out[5]: 1

A função `find_neighbors` busca os vizinhos de um sudoku. Os vizinhos são basicamente outros sudokus, só que com uma casa preenchida de diferença do sudoku pai. Essa casa é preenchida de forma que não invalida a regra da quantidade de números por linha e coluna.

```
In [6]: neighbors = find_neighbors(sudoku)

for idx, neighbor in enumerate(neighbors):
    print(f"#{idx}:")
    print_sudoku(neighbor)
```

```
#0:
1 _ _ 2
_ 2 _ _
2 4 1 3
3 _ 2 _
#1:
4 _ _ 2
_ 2 _ _
2 4 1 3
3 _ 2 _
```

## Experimentos

Com as funções básicas definidas, podemos aplicar elas nos algoritmos especificados do trabalho, sendo eles: busca em profundidade iterativa, busca A\*, subida de encosta e busca por menor custo.

Em cada solução do algoritmo podemos visualizar algumas informações com a função `generate_output`.

### Busca em profundidade iterativa

```
In [7]: solution, sG = iter_depth_search(initial_state=sudoku, goal_check=goal_check, find_neighb

print("solution: \n")
print_sudoku(solution[-1])
```

solution:

```
1 3 4 2
4 2 3 1
2 4 1 3
3 1 2 4
```

```
In [8]: output_gen.generate_output((solution,sG), filename="sudoku_iter_depth", calculate_soluti
```

```
Caminho da solução escrito em: /home/bufulin/Desktop/IA-trabalho-1/relatorio/sudoku_ite
r_depth
Número de nós visitados: 68
Total de nós: 68
Profundidade da meta: 8
Custo da solução: 9
```

```
Out[8]: (68, 68, 8, 9)
```

## Busca A\*

Veja como o A\* sa bem menos nós que a busca em profundidade iterativa. Mas à partir daqui os resultados começam a ser bem similares

```
In [9]: solution,sG = Astar(initial_state=sudoku, goal_check=goal_check, find_neighbors=find_nei
output_gen.generate_output((solution,sG), filename="sudoku_Astar", calculate_solution_co
```

```
Caminho da solução escrito em: /home/bufulin/Desktop/IA-trabalho-1/relatorio/sudoku_Ast
ar
Número de nós visitados: 14
Total de nós: 15
Profundidade da meta: 8
Custo da solução: 9
```

```
Out[9]: (14, 15, 8, 9)
```

## Busca por subida de encosta

```
In [10]: solution,sG = hill_climbing(initial_state=sudoku, goal_check=goal_check, find_neighbors=
output_gen.generate_output((solution,sG), filename="sudoku_hill_climb", calculate_soluti
```

```
Caminho da solução escrito em: /home/bufulin/Desktop/IA-trabalho-1/relatorio/sudoku_hil
l_climb
Número de nós visitados: 14
Total de nós: 15
Profundidade da meta: 8
Custo da solução: 9
```

```
Out[10]: (14, 15, 8, 9)
```

## Busca por menos custo

```
In [11]: solution,sG = least_cost.least_cost_path(initial_state=sudoku, goal_check=goal_check, fi
output_gen.generate_output((solution,sG), filename="sudoku_hill_climb", calculate_soluti
```

```
Caminho da solução escrito em: /home/bufulin/Desktop/IA-trabalho-1/relatorio/sudoku_hil
l_climb
Número de nós visitados: 15
Total de nós: 15
Profundidade da meta: 8
Custo da solução: 9
```

```
Out[11]: (15, 15, 8, 9)
```

## Bônus

Por acaso, foi feita uma função geradora de sudokus. Então podemos gerar um monte de sudokus e fazer gráficos para comparar seus resultados Em larga escala.

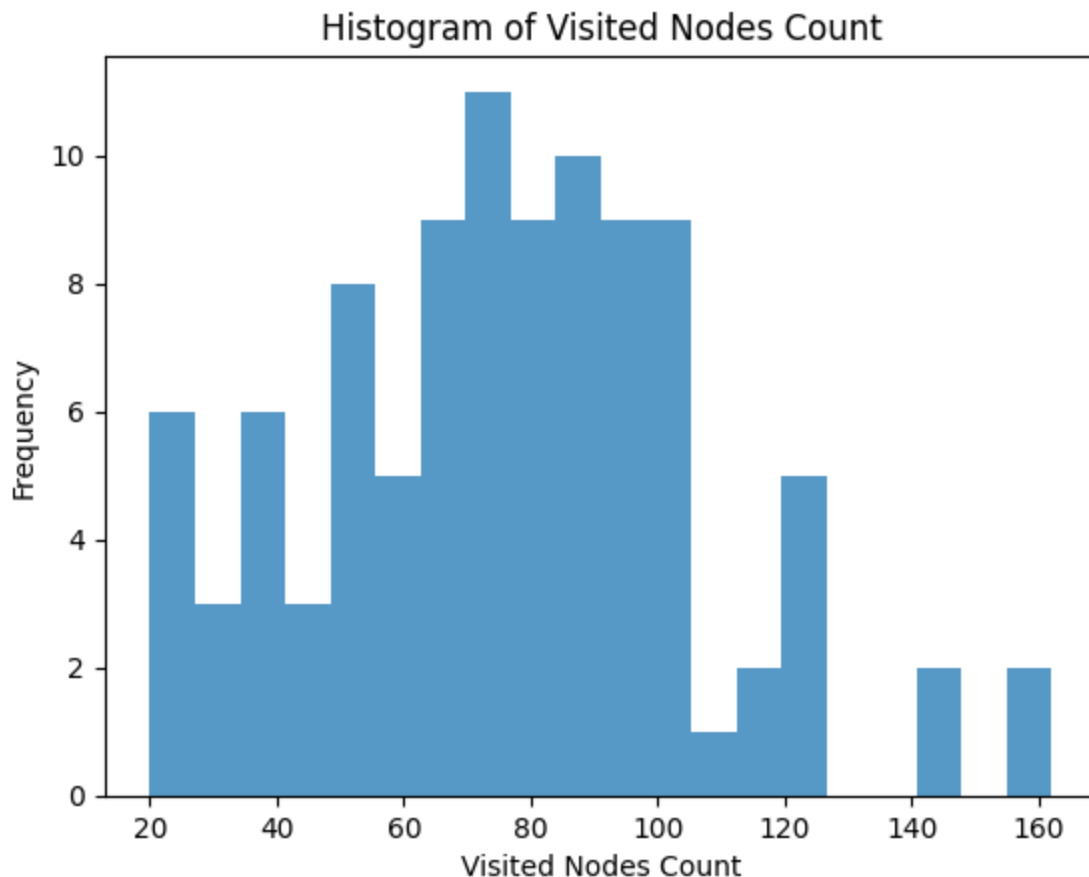
### Múltiplas gerações para o hill climbing

```
In [12]: solutions = []
for i in range(100):
    generated=(create_sudoku_puzzle(4,12))
    solutions.append(hill_climbing(initial_state=generated, goal_check=goal_check, find_
```

```
In [13]: from matplotlib import pyplot as plt

visited_nodes_counts = [output_gen.generate_output((solution[0], solution[1]), calculate

plt.hist(visited_nodes_counts, bins=20, alpha=0.75)
plt.title('Histogram of Visited Nodes Count')
plt.xlabel('Visited Nodes Count')
plt.ylabel('Frequency')
plt.show()
```



```
In [14]: # Data setup
visited_nodes_counts = []
total_nodes_counts = []
goal_depths = []
solution_costs = []

# Iterate over the solutions and unpack the tuples to populate the lists
for solution in solutions:
    visited_nodes_count, total_nodes_count, goal_depth, solution_cost = output_gen.generate(
        (solution[0], solution[1]),
        calculate_solution_cost=calculate_solution_cost
    )

    visited_nodes_counts.append(visited_nodes_count)
    total_nodes_counts.append(total_nodes_count)
    goal_depths.append(goal_depth)
    solution_costs.append(solution_cost)
labels = ['Visited Nodes', 'Total Nodes', 'Goal Depth', 'Solution Cost']

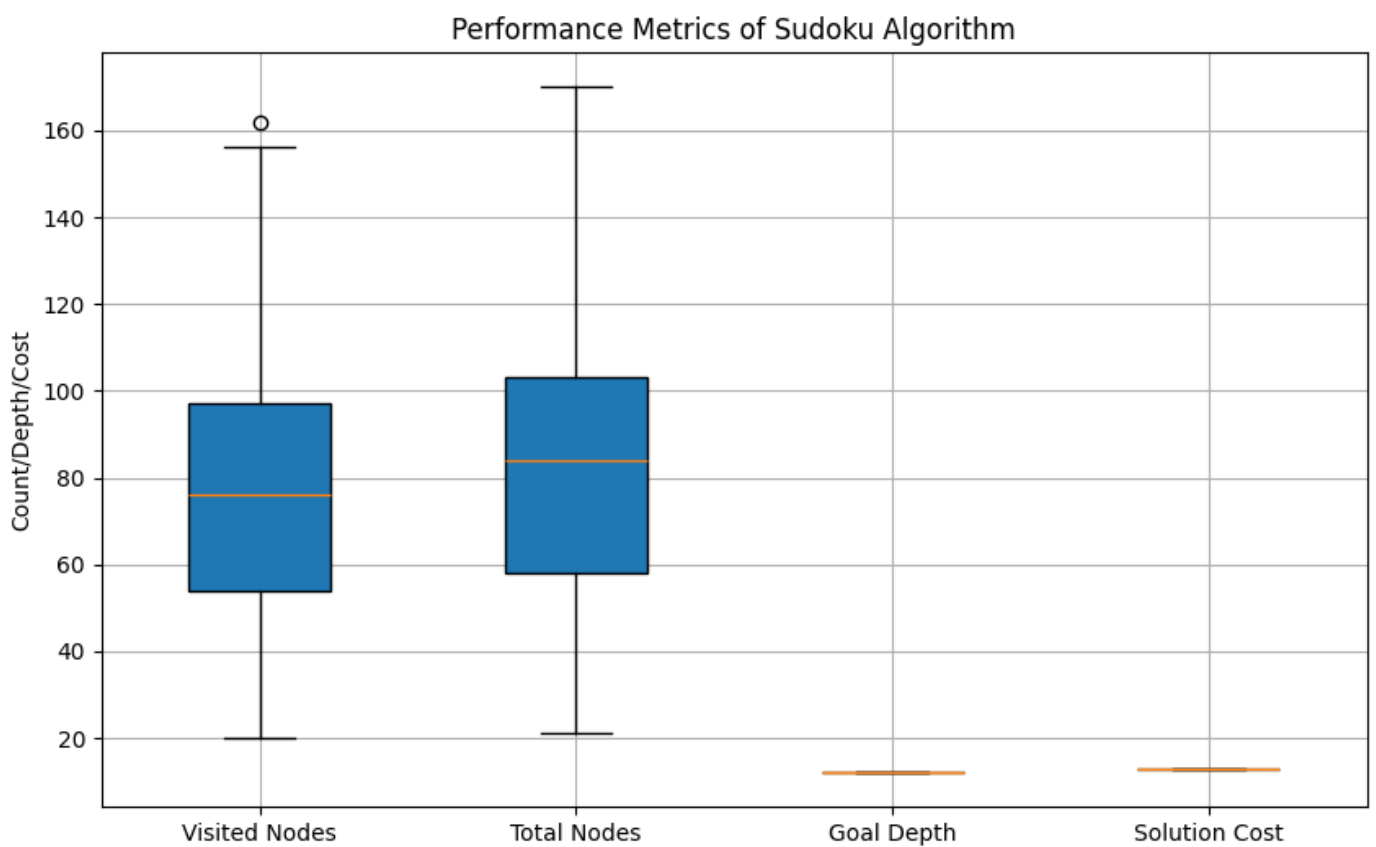
data = [visited_nodes_counts, total_nodes_counts, goal_depths, solution_costs]

# Box plot
plt.figure(figsize=(10, 6)) # You can adjust the size as needed
plt.boxplot(data, labels=labels, patch_artist=True)

# Adding titles and labels
plt.title('Performance Metrics of Sudoku Algorithm')
plt.ylabel('Count/Depth/Cost')

# Adding grid for better readability
plt.grid(True)

# Show the plot
plt.show()
```



```
In [15]: labels = ['Visited Nodes', 'Total Nodes', 'Goal Depth', 'Solution Cost']

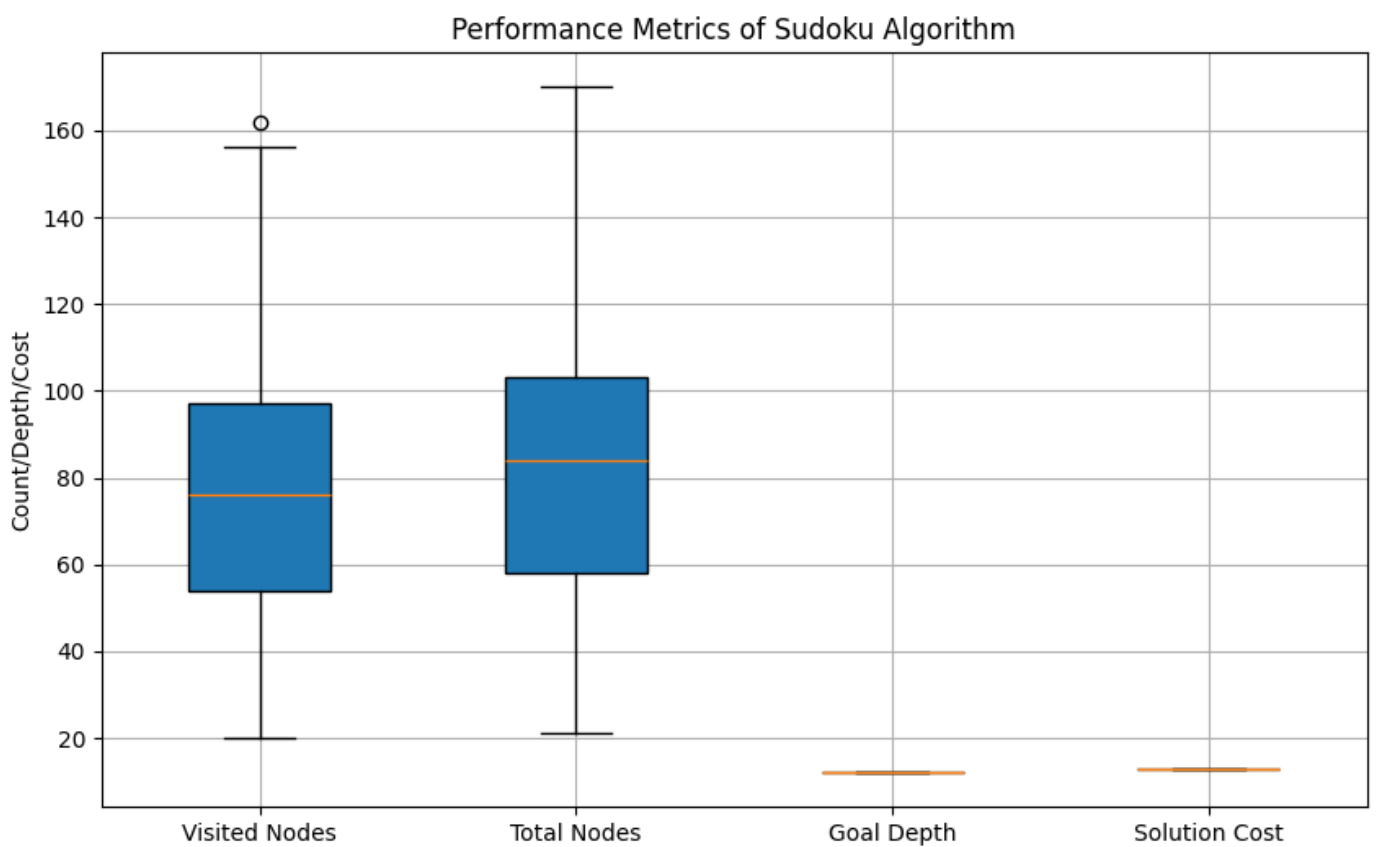
data = [visited_nodes_counts, total_nodes_counts, goal_depths, solution_costs]

# Box plot
plt.figure(figsize=(10, 6)) # You can adjust the size as needed
plt.boxplot(data, labels=labels, patch_artist=True)

# Adding titles and labels
plt.title('Performance Metrics of Sudoku Algorithm')
plt.ylabel('Count/Depth/Cost')

# Adding grid for better readability
plt.grid(True)

# Show the plot
plt.show()
```



Múltiplas gerações para o Astar

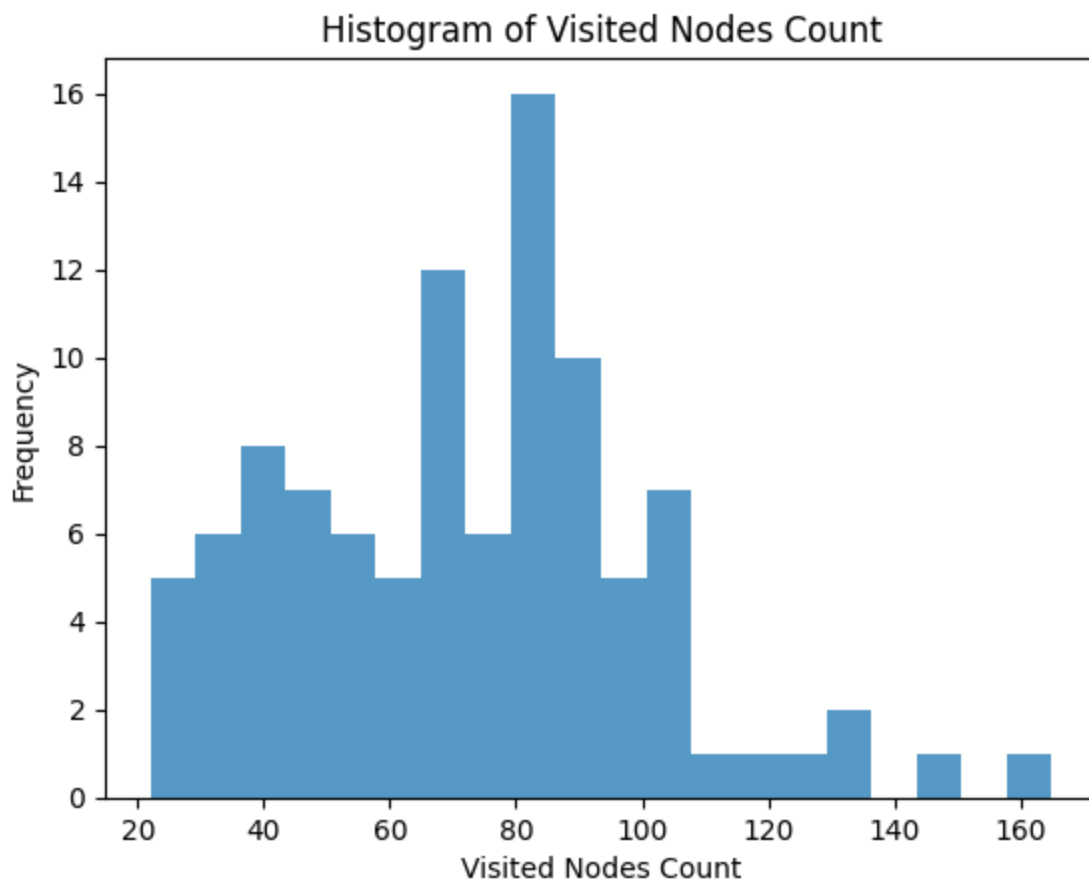
```
In [16]: solutions = []
for i in range(100):
    generated=(create_sudoku_puzzle(4,12))
    solutions.append(Astar(initial_state=generated, goal_check=goal_check,
                           find_neighbors=find_neighbors,
                           heuristic=heuristic,
                           cost_between=cost_between))
```

```
In [17]: from matplotlib import pyplot as plt

visited_nodes_counts = [output_gen.generate_output((solution[0], solution[1]), calculate

plt.hist(visited_nodes_counts, bins=20, alpha=0.75)
plt.title('Histogram of Visited Nodes Count')
plt.xlabel('Visited Nodes Count')
plt.ylabel('Frequency')
plt.show()
```





```
In [18]: # Data setup
visited_nodes_counts = []
total_nodes_counts = []
goal_depths = []
solution_costs = []

# Iterate over the solutions and unpack the tuples to populate the lists
for solution in solutions:
    visited_nodes_count, total_nodes_count, goal_depth, solution_cost = output_gen.generate(
        (solution[0], solution[1]),
        calculate_solution_cost=calculate_solution_cost
    )

    visited_nodes_counts.append(visited_nodes_count)
    total_nodes_counts.append(total_nodes_count)
    goal_depths.append(goal_depth)
    solution_costs.append(solution_cost)
labels = ['Visited Nodes', 'Total Nodes', 'Goal Depth', 'Solution Cost']

data = [visited_nodes_counts, total_nodes_counts, goal_depths, solution_costs]

# Box plot
plt.figure(figsize=(10, 6)) # You can adjust the size as needed
plt.boxplot(data, labels=labels, patch_artist=True)

# Adding titles and labels
plt.title('Performance Metrics of Sudoku Algorithm')
plt.ylabel('Count/Depth/Cost')

# Adding grid for better readability
plt.grid(True)
```

```
# Show the plot  
plt.show()
```

