Resolvendo Sudoku com Busca em Espaço de Estados

Gabriel Lima Nunes[1]

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

gabrielliman@ufmg.br

Resumo. *Este trabalho investiga a eficácia de diversos algoritmos de busca na resolução de problemas de Sudoku. Foram implementados diferentes algoritmos, cada um utilizando estruturas de dados específicas e heurísticas para melhorar o desempenho. A documentação descreve sucintamente o funcionamento dos algoritmos, incluindo a modelagem do problema e os componentes da busca. Heurísticas selecionadas são explicadas brevemente, com justificativas para sua escolha. Análises quantitativas para níveis de dificuldade variados do Sudoku são apresentadas, com tabelas comparativas do número de estados expandidos e do tempo de execução. Os resultados são discutidos, destacando os pontos fortes e limitações de cada algoritmo. Este estudo fornece insights sobre a eficiência dos algoritmos de busca na resolução de Sudoku e suas implicações práticas.*

1. Introdução

O Sudoku é um quebra-cabeça lógico que ganhou popularidade mundial devido à sua simplicidade de regras e desafio de solução. Consiste em preencher uma grade 9x9 com números de 1 a 9, de modo que cada linha, coluna e subgrade 3x3 contenha todos os dígitos de 1 a 9 sem repetição. Apesar de sua simplicidade aparente, o Sudoku apresenta um desafio computacional significativo devido à sua natureza de busca em um espaço de estados enorme.

Neste trabalho, exploramos a eficácia de diversos algoritmos de busca na resolução de problemas de Sudoku. O objetivo é encontrar soluções eficientes que possam resolver Sudokus de diferentes níveis de dificuldade, desde os mais simples até os mais desafiadores. Para isso, implementamos uma variedade de algoritmos de busca, cada um com suas próprias estratégias e heurísticas.

A documentação detalha o funcionamento dos algoritmos, incluindo a modelagem do problema e os componentes da busca, como estado, função sucessora e função verificadora. Também descrevemos as heurísticas escolhidas e justificamos suas seleções com base em suas capacidades de acelerar a busca por soluções viáveis.

2. Funções Auxiliares:

Nesta seção, são apresentadas e explicadas as funções auxiliares utilizadas para o problema do Sudoku.

Função `verificar_objetivo(matriz)`:

Esta função verifica se o estado atual do Sudoku representa uma solução válida. Ela percorre cada linha, coluna e subgrade 3x3 da matriz do Sudoku, verificando se cada

um contém números únicos de 1 a 9. Se algum desses elementos não atender a essa condição, a função retorna `False`, indicando que o estado não é uma solução válida. Caso contrário, retorna `True`.

Função `gerar_sucessores(matriz)`:

Esta função gera os sucessores válidos do estado atual do Sudoku. Ela itera sobre cada célula vazia na matriz, preenchendo-a com números válidos de 1 a 9 e verificando se o estado resultante é válido. Se for válido, adiciona-o à lista de sucessores. A função retorna uma lista contendo todos os sucessores válidos.

Função `validar_atribuicao(matriz, linha, coluna, num)`:

Esta função verifica se é possível atribuir o número `num` à célula na posição (`linha`, `coluna`) da matriz do Sudoku. Ela verifica se o número já está presente na linha, na coluna ou na subgrade 3x3 à qual a célula pertence. Se o número já estiver presente em algum desses locais, a função retorna `False`, indicando que a atribuição não é válida. Caso contrário, retorna `True`, indicando que a atribuição é válida.

3. Breadth-First Search (BFS):

O algoritmo BFS é um método sistemático de busca em árvore ou grafo, onde a exploração é feita nível por nível, começando pelo nó inicial. Ele mantém uma fila de nós a serem explorados e sempre explora o nó mais próximo do nó inicial antes de prosseguir para os nós mais distantes.

Estruturas de dados utilizadas:

- Deque (fila duplamente terminada): Utilizada para armazenar os estados a serem explorados, permitindo adicionar novos estados à direita e remover estados explorados à esquerda.

Implementação do algoritmo:

1. O algoritmo começa adicionando o estado inicial (Sudoku inicial) à fila de estados a serem explorados.
2. Em um loop, ele continua removendo o primeiro estado da fila e verificando se é a solução.
3. Se o estado não for a solução, são gerados seus sucessores válidos e adicionados à fila.
4. O processo continua até que a solução seja encontrada ou a fila de estados a serem explorados esteja vazia.

4. Iterative Deepening Search (IDS):

O Iterative Deepening Search (IDS) é uma técnica de busca que combina as vantagens da busca em profundidade com a garantia de que nenhum nó será visitado mais de uma vez. Ele realiza várias iterações de busca em profundidade com limites crescentes de profundidade até encontrar uma solução. Dessa forma, ele aproveita a eficiência da busca em profundidade para alcançar a profundidade desejada e também garante a completude da busca.

Algorithm 1 Breadth-First Search (BFS) for Sudoku

Input: Initial Sudoku state $sudoku\_initial$
Output: Solution Sudoku state $solution$ or None if no solution is found
$queue \leftarrow$ Empty deque
$num\_visited \leftarrow 0$
$queue$.append($sudoku\_initial$)
while $queue$ is not empty **do**
    $state \leftarrow queue$.pop_left()
    $num\_visited \leftarrow num\_visited + 1$
    if verify_goal($state$) **then**
        return $state$, $num\_visited$
    end if
    $successors \leftarrow$ generate_successors($state$)
    for $succ$ in $successors$ **do**
        $queue$.append($succ$)
    end for
end while
return None, $num\_visited$

Implementação do algoritmo:

1. O algoritmo começa com uma profundidade máxima inicial igual a 1.
2. Realiza uma busca em profundidade limitada com a profundidade máxima atual.
3. Se uma solução é encontrada, retorna a solução.
4. Caso contrário, incrementa a profundidade máxima e repete o processo.
5. Continua aumentando a profundidade máxima até encontrar uma solução ou atingir um limite predefinido.

Algorithm 2 Iterative Deepening Search (IDS) for Sudoku

Input: Initial Sudoku state $sudoku\_initial$
Output: Solution Sudoku state $solution$ or None if no solution is found
$depth\_limit \leftarrow 1$
$num\_visited \leftarrow 0$
$depth\_limit = 1$
while True **do**
    $solution, num\_visited\_lim \leftarrow$ DFS_limited($sudoku\_initial$, $depth\_limit$)
    $num\_visited \leftarrow num\_visited + num\_visited\_lim$
    if $solution$ is not None **then**
        return $solution$, $num\_visited$
    else
        $depth\_limit \leftarrow depth\_limit + 1$
    end if
end while

A função `dfs_limitado` implementa a busca em profundidade (DFS) com limite de profundidade máxima para resolver o problema do Sudoku. A busca em profundidade é realizada de forma recursiva, explorando os sucessores de cada estado até atingir a profundidade máxima especificada.

Algorithm 3 Depth-First Search (DFS) Limitado para Sudoku

Input: Estado inicial do Sudoku $sudoku$, Profundidade máxima $profundidade\_maxima$
Output: Solução do Sudoku $solution$ ou None se nenhuma solução for encontrada

Function DFS_RECURSIVO($estado$, $profundidade$):
if $profundidade > profundidade\_maxima$ **then**
  return None, 0 {Retorna None se a profundidade máxima foi atingida}
end if
if verificar_objetivo($estado$) **then**
  return $estado$, 1 {Retorna o estado se for o objetivo}
end if
$sucessores \leftarrow$ gerar_sucessores($estado$)
$num\_visitados \leftarrow 1$ {Contador de estados visitados}
for each suc **in** $sucessores$ **do**
  $solucao, visitados \leftarrow$ DFS_RECURSIVO(suc, $profundidade + 1$)
  $num\_visitados \leftarrow num\_visitados + visitados$ {Atualiza o contador de estados visitados}
  if $solucao$ **is not** None **then**
    return $solucao, num\_visitados$ {Retorna a solução se encontrada}
  end if
end for
return None, $num\_visitados$ {Retorna None se não encontrar uma solução nesta ramificação}

End Function

A função DFS_RECURSIVO é uma função auxiliar que executa a busca em profundidade de forma recursiva, verificando se a profundidade máxima foi atingida e se o estado atual é a solução. Em seguida, ela explora os sucessores do estado atual até encontrar uma solução ou atingir o limite de profundidade. O número total de estados visitados é retornado como parte do resultado.

5. Uniform Cost Search (Busca de Custo Uniforme):

A Busca de Custo Uniforme é um algoritmo de busca sem informação que expande o nó com o menor custo de caminho conhecido. Ele mantém uma fila de prioridade ordenada pelo custo acumulado de cada nó, de forma que os nós com menor custo são explorados primeiro. Uma vez que o custo utilizado é o numero de posições ainda não preenchidas a busca irá se assemelhar a um DFS.

Funcionamento do algoritmo:

O algoritmo começa com uma fila de prioridade inicialmente contendo o estado inicial com custo zero. Em cada iteração, ele remove o estado com o menor custo da fila de prioridade e o expande, gerando seus sucessores e adicionando-os à fila com seus custos atualizados. Esse processo continua até que o estado objetivo seja encontrado ou até que a fila de prioridade esteja vazia.

Estruturas de dados utilizadas:

- Fila de prioridade: Utilizada para armazenar os estados a serem explorados, ordenados pelo custo acumulado.

Implementação do algoritmo:

O código abaixo mostra a implementação da Busca de Custo Uniforme para resolver o problema do Sudoku.

Algorithm 4 Uniform Cost Search (Busca de Custo Uniforme) para Sudoku

Input: Estado inicial do Sudoku $sudoku\_inicial$
Output: Solução do Sudoku $solution$ ou None se nenhuma solução for encontrada

$fila \leftarrow [(0, sudoku\_inicial.tobytes())]$ {Fila de prioridade inicialmente contendo o estado inicial com custo zero}
$visitados \leftarrow 0$ {Contador de estados visitados}
while $fila$ não estiver vazia **do**
    $custo, estado \leftarrow$ heapq.heappop($fila$) {Remove e retorna o estado com menor custo da fila}
    $estado \leftarrow$ np.frombuffer($estado$, dtype=int).reshape($(9, 9)$) {Convertendo a string de volta para a matriz}
    $visitados \leftarrow visitados + 1$ {Incrementa o contador de estados visitados}
    if verificar_objetivo($estado$) **then**
        return $estado, visitados$ {Retorna o estado e o número de estados visitados se for o objetivo}
    end if
    $sucessores \leftarrow$ gerar_sucessores($estado$)
    for each suc **in** $sucessores$ **do**
        $estado\_str \leftarrow suc$.tobytes() {Convertendo a matriz em uma string}
        heapq.heappush($fila$, (np.count_nonzero($suc == 0$), $estado\_str$)) {Adiciona o sucessor à fila com seu custo}
    end for
end while
return None, $visitados$

6. Busca Gulosa (Greedy Search):

A Busca Gulosa é um algoritmo de busca heurística que seleciona o próximo estado a ser explorado com base em uma heurística de avaliação. Neste contexto, a heurística é escolhida para priorizar estados que parecem mais promissores em direção à solução, sem garantir a otimalidade.

Funcionamento do algoritmo:

O algoritmo de busca gulosa para o Sudoku funciona selecionando o próximo estado com base em uma heurística que avalia o quão promissor é cada estado em direção à solução. Ele escolhe o estado que parece mais próximo da solução de acordo com a heurística utilizada e continua expandindo esse estado até que a solução seja encontrada ou até que não haja mais estados a serem explorados.

Estruturas de dados utilizadas:

- Fila de prioridade: Utilizada para armazenar os estados a serem explorados, ordenados pela avaliação da heurística. Isso permite que o algoritmo selecione o próximo estado de acordo com a heurística utilizada.

Heurística utilizada:

A heurística utilizada neste algoritmo foi a heurística de análise de posição com menos valores possíveis, de modo que a chance de erro ao escolher um valor para preencher uma posição seja minimizado, diminuindo o número de estados a serem explorados

Pseudocódigo do algoritmo:

Algorithm 5 Busca Gulosa para Sudoku

Input: Estado inicial do Sudoku $sudoku\_inicial$
Output: Solução do Sudoku $solution$ ou None se nenhuma solução for encontrada

$fila\_prioridade \leftarrow []$ {Fila de prioridade inicialmente vazia}
avaliacao $\leftarrow$ calcular_avaliacao($sudoku\_inicial$) {Calcular avaliação da heurística para o estado inicial}
insere_fila_prioridade($fila\_prioridade, sudoku\_inicial,$ avaliacao) {Insere o estado inicial na fila de prioridade}
while $fila\_prioridade$ não estiver vazia **do**
    $estado \leftarrow$ remove_melhor_estado($fila\_prioridade$) {Remove e retorna o estado com a melhor avaliação da fila de prioridade}
    if verificar_objetivo($estado$) **then**
        return $estado$ {Retorna o estado se for a solução}
    end if
    $sucessores \leftarrow$ gerar_sucessores($estado$)
    for each suc **in** $sucessores$ **do**
        avaliacao $\leftarrow$ calcular_avaliacao(suc) {Calcular avaliação da heurística para o sucessor}
        insere_fila_prioridade($fila\_prioridade, suc,$ avaliacao) {Insere o sucessor na fila de prioridade}
    end for
end while
return None {Retorna None se nenhuma solução for encontrada}

7. Algoritmo A* para Resolver Sudoku:

O algoritmo A* é um método de busca informada que combina a eficiência da busca gulosa com a garantia de encontrar uma solução ótima, desde que determinadas condições sejam satisfeitas. Ele utiliza uma função heurística para estimar o custo para alcançar a solução a partir de cada estado.

Funcionamento do algoritmo:

O algoritmo A* para o Sudoku utiliza uma função heurística para avaliar o custo total estimado para alcançar a solução a partir de cada estado. Ele mantém uma fila de prioridade ordenada pelo custo acumulado até o momento mais a estimativa heurística para a solução. O algoritmo continua expandindo o estado com o menor custo total até encontrar a solução ou até que não haja mais estados a serem explorados.

Estruturas de dados utilizadas:

- Fila de prioridade: Utilizada para armazenar os estados a serem explorados, ordenados pelo custo total acumulado mais a estimativa heurística. Isso permite que o algoritmo escolha o próximo estado com base em sua prioridade na fila.

Heurística utilizada:

A heurística utilizada neste algoritmo é a heurística de Manhattan, que avalia um estado com base na distância Manhattan de cada célula vazia à célula central do sudoku, assumindo que posições preenchidas mais próximas do centro podem afetar mais o resultado final. Somando esse valor ao número de valores possíveis para preencher essa célula.

Pseudocódigo do algoritmo:

O pseudocógigo se encontra no Algoritmo 6:

8. Resultados

8.1. BFS

Apesar de ser uma busca sem informação em que não utilizamos muitas das propriedades de um tabuleiro do sudoku ao nosso favor, testando somente se uma atribuição é possível antes de realizá-la, o resultado em questão de tempo foi razoável nos casos mais fáceis e intermediários. Entretando no caso mais difícil vemos um número muito alto de estados expandidos e um tempo de execução também alto, isso se deve ao custo de tempo e de espaço serem $O(b^d)$, o que causa aumentos exponenciais com o aumento do número de espaços a serem preenchidos.

Sudoku Extra-Fácil:

Estados expandidos: 313 Tempo de execução (ms): 45

Solução: 137986452 925347168 864521973 753814629 612739845 489652317 571493286 298165734 346278591

Sudoku Intermediário:

Estados expandidos: 4632 Tempo de execução (ms): 604

Solução: 534678912 672195348 198342567 859761423 426853791 713924856 961537284 287419635 345286179

Sudoku Difícil:

Estados expandidos: 2068781 Tempo de execução (ms): 294607

Solução: 812753649 943682175 675491283 154237896 369845721 287169534 521974368 438526917 796318452

Algorithm 6 Algoritmo A* para Sudoku

Input: Estado inicial do Sudoku $sudoku\_inicial$
Output: Solução do Sudoku $solution$ ou None se nenhuma solução for encontrada

$visitados \leftarrow 0$
$heap \leftarrow []$ {Fila de prioridade inicialmente vazia}
heuristica $\leftarrow$ heuristica_manhattan$(sudoku_inicial)$
insere_fila_prioridade$(heap, (0, 0, sudoku_inicial,$ heuristica$))$ {Adicionando o custo acumulado como prioridade}
while $heap$ não estiver vazia **do**
    , custo_acumulado, $sudoku$, heuristica $\leftarrow$ remove_melhor_estado$(heap)$
    $sudoku \leftarrow$ np.frombuffer$(sudoku, dtype = int).reshape((9, 9))$
    $visitados \leftarrow visitados + 1$
    proxima_jogada $\leftarrow$ buscar_proxima_jogada_Astar$(sudoku,$ heuristica$)$
    if proxima_jogada $=$ None **then**
        return $sudoku$, $visitados$ {Retorna o estado se for a solução}
    end if
    $i, j \leftarrow$ proxima_jogada
    $valores\_possiveis \leftarrow$ Atribuições possíveis para posição
    if len$(valores_possiveis) = 0$ **then**
        {Pula para a próxima iteração se não houver valores possíveis}
    end if
    for each $valor$ **in** $valores\_possiveis$ **do**
        $novo\_sudoku \leftarrow$ sudoku.copy()
        $novo\_sudoku[i][j] \leftarrow valor$
        $novo\_custo\_acumulado \leftarrow$ custo_acumulado
        insere_fila_prioridade$(heap, (custo_acumulado, novo_custo_acumulado,$ $novo\_sudoku, valor\_heuristica))$
    end for
end while
return $sudoku$, $visitados$ {Retorna o estado e o número de estados visitados}

Sudoku Intermediário:

Estados expandidos: 27948 Tempo de execução (ms): 3046

Solução: 812753649 943682175 675491283 154237896 369845721 287169534 521974368 438526917 796318452

8.2. IDS

O IDS apresentou os piores resultados gerais para a resolução do problema, uma vez que só é possível achar um estado final válido quando todas as posições estiverem preenchidas todas as buscas anteriores a busca com profundidade máxima igual ao número de espaços vazios se tornam inúteis.

Desse modo, até mesmo nos casos mais fáceis vemos um número de estados expandidos e um custo de tempo muito superior a média, a vantagem desse método de nos garantir uma solução ótima para o problema acaba não se aplicando para o preenchimento do sudoku, mostrando que esse método não é uma escolha adequada.

O custo de tempo do algoritmo é $O(b^d)$, igual ao BFS, mas o de espaço é inferior sendo somente $O(b * d)$.

Sudoku Extra-Fácil:

Estados expandidos: 8917 Tempo de execução (ms): 1126

Solução: 137986452 925347168 864521973 753814629 612739845 489652317 571493286 298165734 346278591

Sudoku Intermediário:

Estados expandidos: 146470 Tempo de execução (ms): 16771

Solução: 534678912 672195348 198342567 859761423 426853791 713924856 961537284 287419635 345286179

Sudoku Difícil:

Estados expandidos: 64395813 Tempo de execução (ms): 12555761

Solução: 812753649 943682175 675491283 154237896 369845721 287169534 521974368 438526917 796318452

Sudoku Intermediário:

Estados expandidos: 864795 Tempo de execução (ms): 59779

Solução: 812753649 943682175 675491283 154237896 369845721 287169534 521974368 438526917 796318452

8.3. Uniform Cost Search:

O Uniform Cost Search apresentou os melhores resultados entre os algoritmos sem informação, tendo sua lógica de execução semelhante a um DFS um vez que a função de custo escolhida é a contagem do número de zeros na matriz, desse modo o algoritmo tentará se aprofundar o máximo possível, preenchendo posições do sudoku até resultar em erro e realizando backtracking.

O custo de tempo do algoritmo é $O(b^d)$ no pior caso mas no caso médio efetivamente é muito menor, o que pode ser notado pelo número de estados expandidos, e o custo de espaço é $O(b * d)$.

Sudoku Extra-Fácil:

Estados expandidos: 94 Tempo de execução (ms): 17

Solução: 137986452 925347168 864521973 753814629 612739845 489652317 571493286 298165734 346278591

Sudoku Intermediário:

Estados expandidos: 4209 Tempo de execução (ms): 564

Solução: 534678912 672195348 198342567 859761423 426853791 713924856 961537284 287419635 345286179

Sudoku Difícil:

Estados expandidos: 49559 Tempo de execução (ms): 8006

Solução: 812753649 943682175 675491283 154237896 369845721 287169534 521974368 438526917 796318452

Sudoku Intermediário:

Estados expandidos: 1309 Tempo de execução (ms): 179

Solução: 812753649 943682175 675491283 154237896 369845721 287169534 521974368 438526917 796318452

8.4. Algoritmo Guloso:

O Algoritmo Guloso apresentou o melhor resultado entre todos os testes ao utilizar uma heurística que reduziu muito o branching factor efetivo, ao selecionarmos a posição com o menor número de opções possíveis de preenchimento reduzimos muito a chance de erro, mitigando o número de estados gerados. Isso é refletido no menor tempo de execução e menor número de estados entre todos os algoritmos.

Sudoku Extra-Fácil:

Estados expandidos: 47 Tempo de execução (ms): 7

Solução: 137986452 925347168 864521973 753814629 612739845 489652317 571493286 298165734 346278591

Sudoku Intermediário:

Estados expandidos: 52 Tempo de execução (ms): 7

Solução: 534678912 672195348 198342567 859761423 426853791 713924856 961537284 287419635 345286179

Sudoku Difícil:

Estados expandidos: 10102 Tempo de execução (ms): 2240

Solução: 812753649 943682175 675491283 154237896 369845721 287169534 521974368 438526917 796318452

Sudoku Intermediário:

Estados expandidos: 212 Tempo de execução (ms): 49

Solução: 812753649 943682175 675491283 154237896 369845721 287169534 521974368 438526917 796318452

8.5. A*:

O Algoritmo A* mostrou eficiência superior aos algoritmos sem informação mas não alcançou resultados tão bons quando o Algoritmo Guloso, acredito que a diferença em desempenho se deve a escolha de heuristicas diferentes como especificado pelo trabalho. Ambas reduziram o branching factor efetivo mas o desempenho de cada uma foi diferente. Apesar disso, podemos considerar que ambos os algoritmos com informação conseguiram resultados em alta velocidade e com poucos estados.

Sudoku Extra-Fácil:

Estados expandidos: 65 Tempo de execução (ms): 11

Solução: 137986452 925347168 864521973 753814629 612739845 489652317 571493286 298165734 346278591

Sudoku Intermediário:

Estados expandidos: 54 Tempo de execução (ms): 12

Solução: 534678912 672195348 198342567 859761423 426853791 713924856 961537284 287419635 345286179

Sudoku Difícil:

Estados expandidos: 154332 Tempo de execução (ms): 39217

Solução: 812753649 943682175 675491283 154237896 369845721 287169534 521974368 438526917 796318452

Sudoku Intermediário:

Estados expandidos: 7327 Tempo de execução (ms): 1583

Solução: 812753649 943682175 675491283 154237896 369845721 287169534 521974368 438526917 796318452