Resolvendo Sudoku com Busca em Espaço de Estados

Igor Joaquim da Silva Costa[1]

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

igor.joaquim@dcc.ufmg.br

Matrícula - 2021032218

1. Introdução

O paradigma de busca em espaço de estados é uma das ideias mais simples porém poderosas da computação. Com ele, é possível modelar matematicamente um problema e ir, a cada passo, montando configurações diferentes até chegarmos em um ponto de utilidade desejada. Nesse sentido, temos que lidar com duas decisões de algoritmo complementares, como montar as configurações - ou seja, como definir a vizinhança de um nó - e como percorrer o espaço de forma eficiente.

Este projeto implementa um solver para Sudoku usando vários algoritmos de busca em espaço de estados. Os algoritmos implementados incluem Pesquisa Ampla (BFS), Pesquisa Iterativa de Aprofundamento (IDS), Pesquisa de Custo Uniforme (UCS), Pesquisa A* (A_STAR) e Pesquisa Greedy Best-First (GBFS).

2. Arquitetura do Projeto

- Funcionamento dos algoritmos, estruturas de dados utilizadas, modelagem do problema e componentes da busca (estado, função sucessora, função verificadora, etc)

2.1 Visão geral

Embora tenhamos o objetivo de implementar diferentes algoritmos de busca no mesmo programa, o fluxo de execução geral de uma busca é sempre o mesmo:

1. Escolha um estado atual com base em alguma regra
    a. Verifique se esse estado é final, se for, pare
2. Adicione os vizinhos desse estado em uma estrutura de dados
3. Repita enquanto existir algum estado na estrutura
    a. Caso contrário, o problema não possui solução viável

Dessa forma, para implementar tais abstrações, foi seguido um paradigma de programação orientada a objetos (POO), onde existe uma classe abstrata Busca_Base (`Search_Base`) responsável por definir as abstrações por meio de funções. De maneira mais clara, os passos 1, 2 e 3 são implementados por métodos abstratos na classe Base e

cada classe de algoritmo herda da classe base, com a responsabilidade de implementar os respectivos métodos abstratos.

Modelagem do Problema

Antes de descrever como os algoritmos funcionam, é crucial entender como o problema é desenvolvido. Cada nó do espaço de estados é representado pela classe State(`State`), um objeto que possui conhecimento sobre o tabuleiro que ele modela e seus filhos. Um estado ***S*** gera um filho ***S**** seguindo o seguinte procedimento:

1. Escolha uma posição [x,y] tal que o elemento [x,y] em ***S*** esteja vazio
2. Escolha um valor entre (1…9) tal que esse valor na posição [x,y] configure um estado viável
3. Duplique todos os elementos de ***S*** em ***S****, com exceção do elemento [x,y] que receberá o valor encontrado.

Dessa forma, ***S*** terá no máximo 9 filhos distintos, que é definido como a vizinhança de ***S***. Além disso, cada estado conhece seu custo e sua distância até o nó inicial.

O objetivo é encontrar o estado ***S*** tal que ***S*** tenha todas as posições preenchidas de forma válida.

Algoritmos

Dada a contextualização acima, os algoritmos são implementados de forma trivial. Por exemplo, o algoritmo BFS faz o passo 1 (Escolha um estado atual com base em alguma regra) e 2 (Adicione os vizinhos desse estado em uma estrutura de dados) usando uma ***fila***, garantindo a propriedade de busca em largura. Ou seja, o passo 1 retorna o primeiro elemento da fila e o passo 2 coloca os estados em ordem.

IDS

Similarmente, o algoritmo IDS só é responsável por manipular uma ***pilha,*** de forma que estados com custo maior que o limite estipulado não sejam passíveis de serem explorados. Isso é feito filtrando os estados que são inseridos na ***pilha.***

Para garantir que a execução seja interativa, a busca da IDS, além de manter apenas estados menores que um custo máximo, é recursiva, ou seja, caso nenhuma solução seja encontrada com o limite L, a busca é refeita com o valor de L aumentado, até encontrar a solução. **Após testes experimentais, o aumento de L foi definido como um crescimento exponencial, com L(t) = 2*L(t-1) e L(0) = 1.** Isso foi feito para diminuir o tempo de execução, tendo como referência estratégia de realocação de vetores em c++, onde, sempre que é preciso remontar uma estrutura do zero (nesse caso uma região de memória, no nosso caso uma árvore de estados) a capacidade máxima da estrutura dobra de tamanho, com o intuito de reconstruir a estrutura o mínimo de vezes possível.

Algoritmos baseados em custo

Todos os algoritmos que precisam ordenar os estados com base em algum critério numérico caem nessa categoria (A* UCS e GFBS). Aqui, a estratégia foi simples. Todas as classes herdam de UCS, uma classe que manipula uma ***fila de prioridade PQ***, implementada como um ***heap*** que sempre se mantém ordenado. O critério de ordenação de estados depende de uma função:

$$f(estado) = g(estado) + h(estado)$$

Simplificando, o custo do estado f() depende do custo atual g() e da heurística h(). A classe UCS define inicialmente g() = distância do estado até o nó inicial e h() = 0, logo, a heurística não possui efeito na ordenação.

Paralelamente, no algoritmo A*, h() é sobrescrito para implementar uma heurística. No caso de GFBS, g() = 0 e h() é outra heurística.

3. Heurísticas

Nessa seção, serão explicadas as heurísticas utilizadas.

A Estrela

Para o algoritmo A*, a heurística pensada usa como conhecimento de mundo o custo real da solução. Para o A* ser ótimo, é necessário uma função h(x) que seja sempre menor que o custo da solução real. Isso é implementado da seguinte forma:

O objetivo é reduzir o custo dos estados mais próximos do estado final. Considere que o estado atual ***S*** foi obtido a partir do estado inicial que é completamente vazio, ou seja, toda combinação de [x.y] está marcada com o valor zero. Seja ***I*** esse estado inicial e ***X*** a solução final. A distância de ***I*** -> ***X*** é exatamente 81, já que são necessários 81 zeros para serem preenchidos. A distância de ***I*** -> ***S*** é a quantidade de valores não zeros que ***S*** possui. Rearranjando os termos:

$$I -> X = I -> S + S -> X$$

$$S -> X = I -> X - I -> S$$

$$S -> X = 81 - (I -> S)$$

Dessa forma, temos que h(S) = 81 - (quantidade de posições não nulas em S), um valor sempre positivo, fazendo assim, com que estados com menos restrições sejam explorados primeiro. Entretanto, como g(S) é função de ***I*** -> ***S,*** foi decidido por manter:

$$h(S) = 2 * (81 - (quantidade\ de\ posições\ não\ nulas\ em\ S))$$

Com o intuito de diminuir a dependência do custo do caminho.

GFBS

Como esse algoritmo não utiliza o custo real, é mais fácil definir heurísticas diferentes para ele. Usando desse fato, a heurística usada pondera duas heurísticas diferentes. É

conhecido que o estado final é um estado viável filho de algum estados com muitas restrições. Dessa forma, queremos dar preferência a estados com mais restrições.

O objetivo pode ser obtido de duas formas: a primeira, considerando o tamanho da vizinhança do nó atual (heurística local), e a segunda considerando que estados mais "profundos" na árvore possuem menos restrições. A heurística escolhida foi a média dos dois, ou seja:

$$2^* \, h(S) \; = |Vizinhança(S)| \; + \; Zeros(S)$$

Durante os testes, foi observado que o efeito da vizinhança perde a utilidade ao aumentar a profundidade da árvore. Dito isso, h(S) foi definido como

$$2 \; * \; h(S) \; = \frac{|Vizinhança(S)|}{Profundidade(S)} + Zeros(S)$$

Em alguns casos, o GFBS soluções mais rapidamente do que o A*.

4. Resultados e Discussão

Com o intuito de estressar o sistema e avaliar o desempenho dos algoritmos, uma série de testes foram feitos. Os casos de testes foram retirados de Mantere et. al. [1], com modificações para adequar a entrada do programa. Os testes foram realizados em paralelo, sem que dois processos acessem a mesma entrada, para evitar condições de corrida. Os testes também foram feitos sem o uso de paralelismo, os resultados continuam os mesmos.

Comparação de performance

A fim de comparar os diferentes algoritmos, foi analisada a quantidade de estados explorados, além do tempo médio gasto em cada problema. As instâncias de testes são classificadas com base na quantidade de zeros existentes na entrada.

O primeiro resultado obtido é a correlação entre a quantidade de zeros e a dificuldade do problema. Isso é visto tanto na figura 1. quanto nas correlações da figura 3. Esse tempo de processamento é inerente ao problema, ou seja, existe um limite para o quanto o algoritmo é capaz de agilizar o processo de encontrar a solução. Portanto, a quantidade de zeros será usada como um sinônimo para a dificuldade do problema.

No que diz o comportamento dos algoritmos em si, vemos 3 grupos principais:

no primeiro, os algoritmos de busca informada (A* e GFBS), no segundo, os algoritmos que testam todas as possibilidades (UCS e BFS) e, por fim, o algoritmo iterativo. Pelo tempo de execução, o IDS apresenta um cenário interessante, ele é o algoritmo mais rápido em instâncias fáceis. Dada sua característica de limitar os estados, quando a solução é rasa, a busca em profundidade do IDS acaba sendo a forma de encontrar a solução mais eficiente. Entretanto, como visto na figura 2, essa facilidade se perde ao precisar reiniciar a busca várias vezes, o que é o caso dos casos mais difíceis, fazendo com que a quantidade de espaços explorados cresça ordem de magnitude mais do que em outros algoritmos.

Outro algoritmo interessante foi o UCS, que, da forma com que foi implementado, explora os mesmos estados de uma BFS, só que com um custo O(nlog(n)) a cada expansão para manter a fila de prioridade ordenada, o que acaba aumentando o tempo de execução.

Por fim, os algoritmos de busca informada vão bem na maioria dos casos, com o A* e o GFBS disputando quem vai melhor em cada caso. Como o GFBS faz um caminho diferente do A*, principalmente nos primeiros estados, existem casos que a busca melhora ou piora o tempo de execução. Entretanto o A* se mostra mais estável.

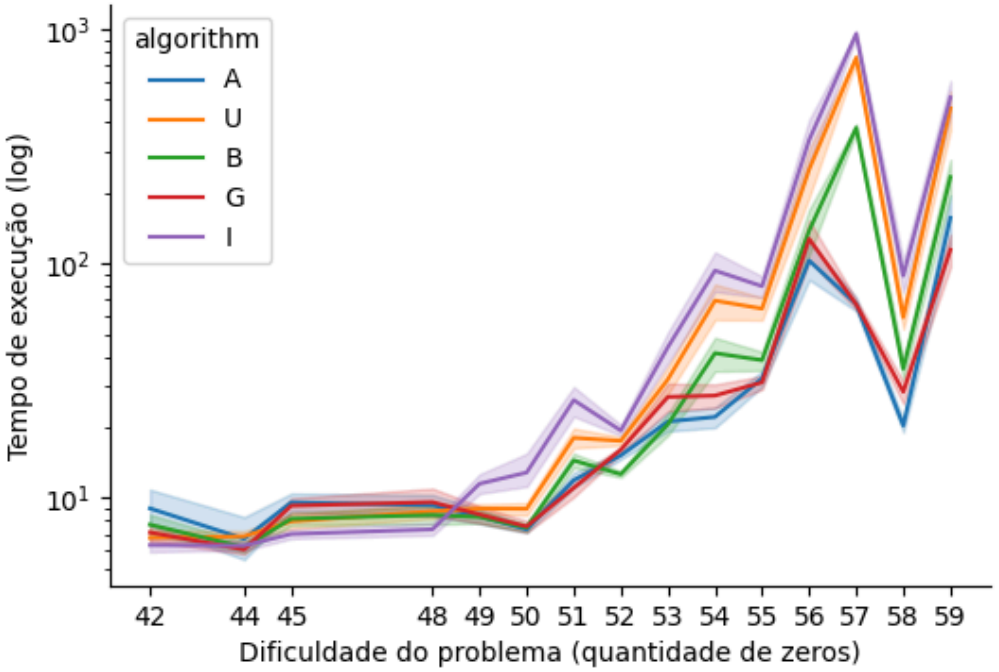

Figura 1. Esse gráfico de linha ilustra a relação entre a dificuldade do problema com o tempo gasto para resolvê-lo. Para qualquer tipo de algoritmo, o tempo aumenta com problemas maiores. Entretanto, alguns algoritmos são mais resilientes.

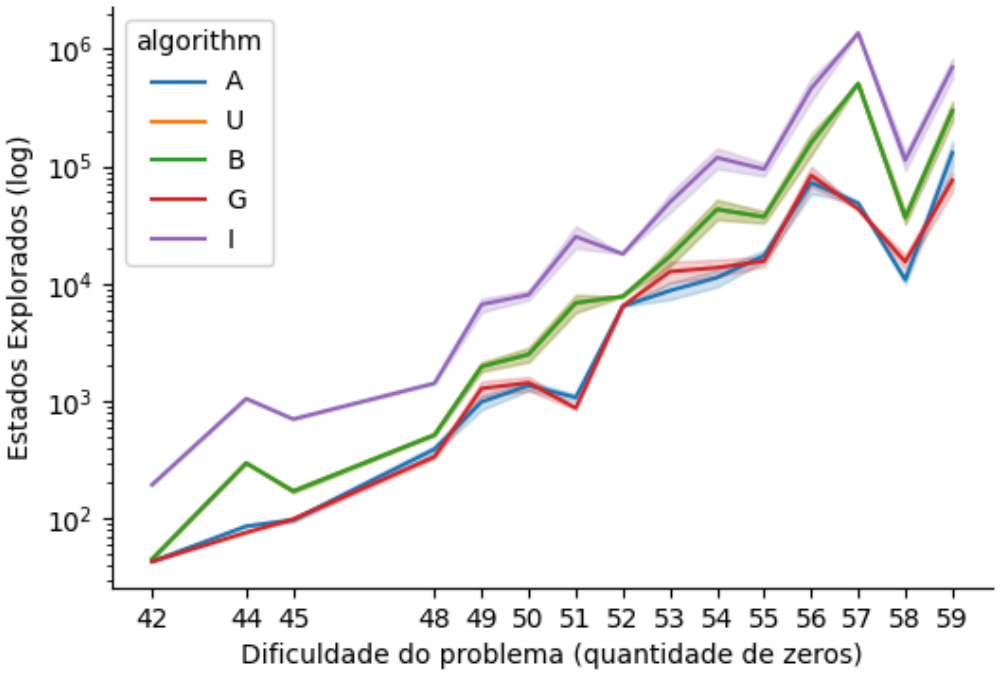

Figura 2. Existe uma variação maior entre os estados explorados e a quantidade de zeros

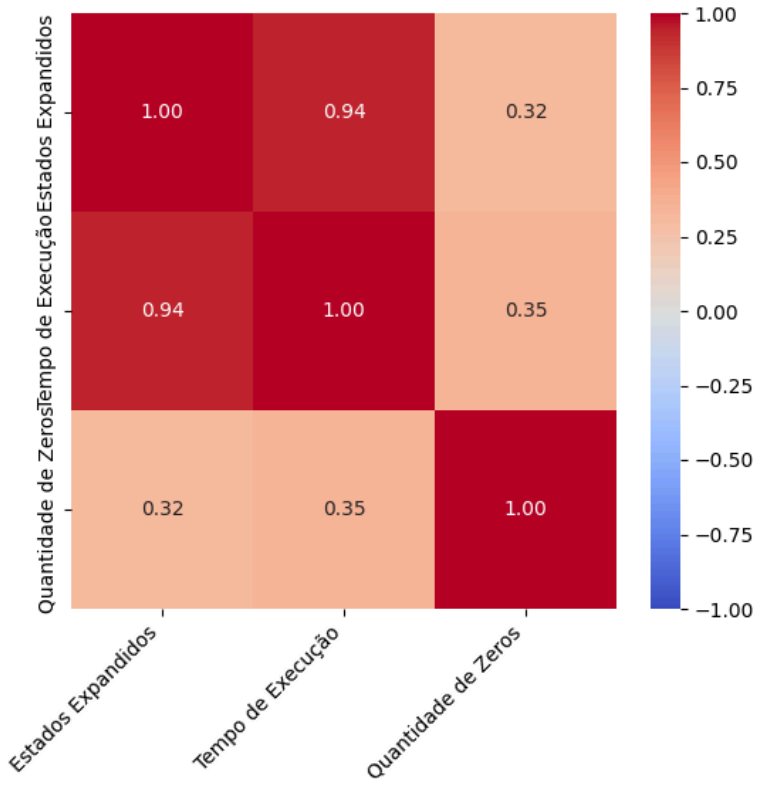

Figura 3. Matriz de Correlação das features.

Critérios de Desempate

Visto que, primeiro, ***existe uma dificuldade inerente ao problema*** e , segundo, ***alguns algoritmos vão melhor do que outros,*** é justo considerar uma forma encontrar o algoritmo que funciona melhor na maioria dos casos. É possível modelar essa resiliência a dificuldade do problema da seguinte forma. Primeiro, o tempo de execução do algoritmo A no problema i depende apenas de dois termos.

$$T(A,i) = N(A) + Dificuldade(i)$$

Onde A é o algoritmo, i é o problema, N(A) é uma variável aleatória com $E[N(A)] = \alpha$ e $Dificuldade(i)$ É uma constante que não depende do algoritmo.

Quando estimamos $E[T(A,i)] \approx \frac{T(A,i)}{\#repetições}$, temos que:

$$E[T(A,i)] = E[N(A)] + E[Dificuldade(i)]$$

$$E[T(A,i)] = \alpha + Dificuldade(i)$$

$$T(A,i) - E[T(A,i)] = T(A,i) - (\alpha + Dificuldade(i))$$

$$T(A,i) - E[T(A,i)] = N(A) + Dificuldade(i) - (\alpha + Dificuldade(i))$$

$$T(A,i) - E[T(A,i)] = N(A) - \alpha$$

Que é uma variável aleatória com média 0, porém independe da dificuldade do problema. Dessa forma, podemos analisar apenas o fator $N(A)$ em ação na figura 4:

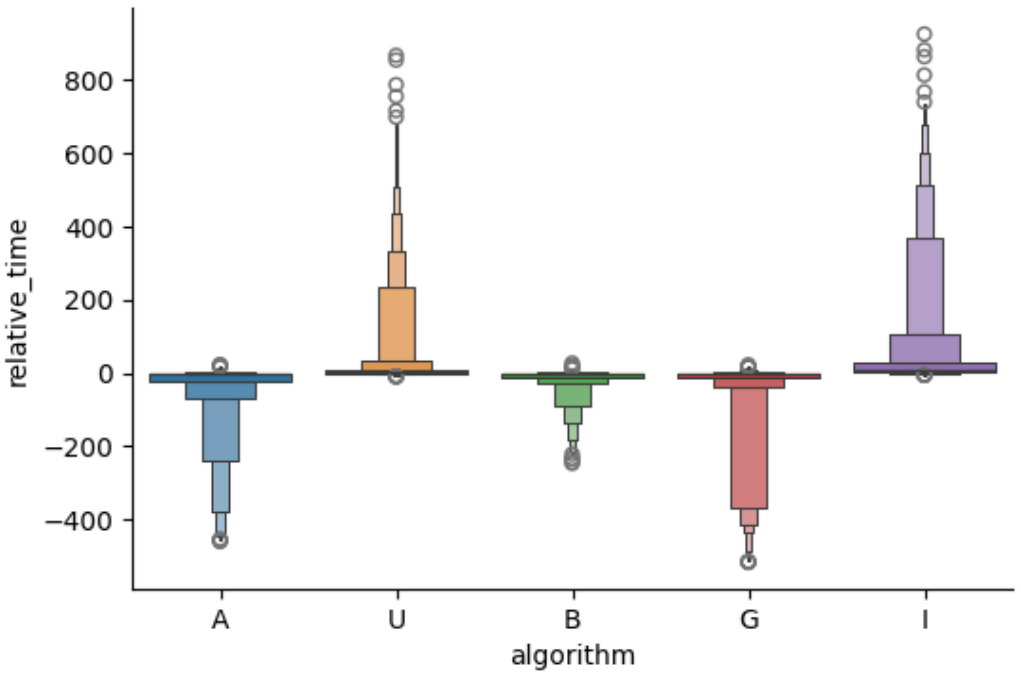

Figura 4. Boxenplot da diferença entre o tempo de execução do algoritmo e o tempo de

execução médio daquela instância. Quanto mais negativo, mais rápido é o algoritmo em relação a média.

Dessa forma, pode-se notar que, em quesito tempo, os algoritmos UCS e IDS foram os piores, já que eles desviam da média positivamente, ou seja, são maiores que a média. Como seus concorrentes são bem ruins, o algoritmo BFS consegue se manter em uma posição segura. Entretanto, nos extremos, o GFBS consegue trazer os melhores resultados em menos tempo, sendo uma escolha viável para resolver o problema de Sudoku

6. Conclusão

O projeto implementou um solver para Sudoku utilizando diversos algoritmos de busca em espaço de estados, como BFS, IDS, UCS, A* e GBFS. Os algoritmos foram organizados dentro de um paradigma de programação orientada a objetos, com uma classe base que define abstrações comuns para todos os algoritmos, enquanto cada classe de algoritmo implementa seus métodos específicos.

Os algoritmos utilizam estruturas de dados apropriadas para garantir eficiência na exploração do espaço de estados, como filas para BFS, pilhas para IDS, e filas de prioridade para algoritmos baseados em custo, como UCS, A*, e GBFS. As heurísticas também foram empregadas nos algoritmos informados (A* e GBFS) para otimizar o processo de busca, com funções específicas projetadas para minimizar o custo do caminho e direcionar a busca de maneira eficaz.

Os resultados de desempenho dos algoritmos foram comparados em diferentes instâncias de Sudoku. O IDS mostrou-se eficiente para instâncias fáceis, enquanto que os algoritmos informados A* e GBFS se saíram melhor em uma variedade de casos.

Os testes evidenciaram que o UCS e o IDS tiveram o pior desempenho em relação à média de tempo de execução, enquanto o GBFS teve os melhores resultados. Em conclusão, o projeto demonstrou a eficácia de aplicar diferentes algoritmos de busca em espaço de estados para resolver Sudoku, e destacou os benefícios e desafios de cada abordagem. Nenhum dos códigos foi devidamente paralelizado, sendo essa uma otimização que configure o próximo passo do projeto

7. Referências

[1] Mantere, Timo and Janne Koljonen (2008). Solving and Analyzing Sudokus with Cultural Algorithms. In Proceedings of 2008 IEEE World Congress on Computational Intelligence (WCCI 2008), 1-6 June, Hong Kong, China, pages 4054-4061

[2] Russell, Stuart J. (Stuart Jonathan), 1962-. Artificial Intelligence : a Modern Approach. Upper Saddle River, N.J. :Prentice Hall, 2010.