Resolvendo Sudoku com Busca em Espaço de Estados
Trabalho Prático - Introdução à Inteligência Artificial

Vinicius Silva Gomes {2021421869}[1]

[1] Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brasil

{vinicius.gomes}@dcc.ufmg.br

1. Introdução

Sudoku é um jogo baseado na colocação lógica de números. O objetivo principal é: dado um tabuleiro 9×9, constituído de 9 regiões 3×3, colocar números de 1 a 9 em cada uma das posições vazias do tabuleiro, de modo que não hajam números repetidos em cada linha, coluna e região do tabuleiro [Wikipedia 2024].

Inicialmente, alguns números são inseridos no tabuleiro, de modo a permitir a indução ou dedução dos números das células não preenchidas. Além disso, os problemas normalmente podem ser classificados de acordo com a quantidade de números presentes na configuração inicial. Quanto menor for o número de posições inicialmente preenchidas, mais difícil será induzir/deduzir o restante das posições.

Por se tratar de um problema NP-completo [Yato and Seta 2003] e, portanto, ser de difícil computação, é interessante estudar a fundo as características particulares do problema e quais abordagens são capazes de solucioná-lo, de modo a perceber as vantagens e desvantagens de cada uma e, com isso, ser capaz de determinar qual a mais adequada para cada padrão diferente de estado inicial do problema, considerando tanto o tempo despendido computando a grade completa quanto a quantidade de memória gasta.

Nesse sentido, esse relatório discute os resultados obtidos a partir da análise de 5 algoritmos de busca diferentes: *Breadth-first Search*, *Uniform-Cost Search*, *Iterative Deepening Search*, *Greedy Best-first Search* e *A* Search*; que exploram o espaço de estados do problema e procuram por uma solução possível. Serão comparados o tempo computacional despendido e o número de nós que foram expandidos até que uma configuração totalmente preenchida fosse encontrada.

A seção 2 discute como os componentes da busca foram definidos para o problema e a 3 como ele foi modelado computacionalmente; na seção 4 a metodologia é apresentada, juntamente com os algoritmos implementados e o processo de experimentação; na seção 5 os resultados obtidos são discutidos; e a seção 6 encerra o relatório.

2. Componentes da busca

Para que seja possível implementar os algoritmos de busca, é necessário entender o problema e mapear as cinco principais componentes que permitem que a busca seja possível: o conjunto de estados e o estado inicial, o conjunto de ações aplicáveis aos estados, a função sucessora, como o teste de objetivo é feito e qual o custo de cada ação.

No mapeamento feito para o Sudoku, o conjunto de estados é composto pelo número de 1 a 9 associado a cada posição na grade do jogo e o estado inicial é informado na entrada do programa.

Quanto as ações, a única ação possível é a de preencher uma célula vazia com um número que não viole as restrições do jogo. A célula escolhida é a primeira encontrada em um percorrimento da grade que se dá de cima para baixo e dá direita para a esquerda. Além disso, o custo de cada ação, ou seja, o custo de preencher uma posição, é um.

Por fim, o teste de objetivo passa quando um estado que não possui nenhuma célula vazia é encontrado. As funções que implementam parcialmente ou totalmente cada um desses componentes serão discutidas a fundo na seção 3.

3. Modelagem computacional

A representação de um estado, foi feita utilizando uma matriz de inteiros preenchidas com zero nas posições que ainda não possuem algum número atribuído. Essa estrutura se mostrou simples e eficiente para a realização de todas as computações que foram necessárias nas implementações dos algoritmos.

Além disso, duas classes principais foram implementadas: `Search` e `Node`. A classe `Search` funciona como um *wrapper* para os algoritmos de busca implementados. Ela tem como único parâmetro a matriz com a configuração inicial do problema, recebida como parâmetro na entrada, e implementa os algoritmos: `breadth_first_search()`, `uniform_cost_search()`, `iterative_deepening_search()`, `greedy_best_first_search()` e `a_star_search()`. Na seção 4, cada algoritmo será melhor detalhado, juntamente com os aspectos pertinentes da implementação feita para cada um deles.

A classe `Node`, por sua vez, é a representação de um nó na árvore de espaço de estados. Sendo assim, uma instância de `Node` armazena sua configuração de estado (grade), sua profundidade na árvore e seu custo. Os métodos implementados pela classe são: `is_goal()`, `count_zeros()`, `count_possibilities()` `is_valid()` e `expand()`.

Os métodos `count_zeros()` e `count_possibilities()` são responsáveis por calcular as informações que serão importantes para as heurísticas adotadas na busca informada. O primeiro conta o número de células que ainda não foram preenchidas em um estado e o segundo percorre a grade e acumula quantos números diferentes podem ser inseridos em cada célula vazia. Ao final ele retorna o número total de possibilidades para a grade inteira.

O método `goal()` implementa uma verificação que testa se o estado daquele nó já atingiu o objetivo da busca, ou seja, se todas as células da grade já foram preenchidas. Para fazer isso, ele percorre a grade e caso uma posição com valor zero seja encontrada, o método retorna o valor booleano *false*. Caso contrário, se nenhuma célula que satisfaça essa condição for encontrada, ele retorna *true*.

A função `is_valid()` é um método auxiliar que recebe como parâmetros três inteiros row, col e $value$ e testa se a posição $[row][col]$ da grade pode ser preenchida com o número $value$. Para tanto, o procedimento percorre a linha, coluna e região associada a posição $[row][col]$ e verifica se alguma célula já foi preenchida com o número $value$. Caso sim, o procedimento retorna o booleano *false* e, caso não, o booleano *true*.

Já o método `expand()`, por fim, expande o nó que invocou a função, gerando uma lista com todos os seus sucessores (filhos). Para fazer isso, esse método, inicial-

mente, identifica uma célula não preenchida. Uma vez que ela tenha sido encontrada, ele inicializa um vetor de instâncias `Node` (vetor de filhos/sucessores) e executa um *loop* de 1 a 9. A cada iteração, o método auxiliar `is_valid()` é chamado recebendo como parâmetro a posição da célula vazia e o valor do iterador. Caso seja válido preencher a célula com esse valor, uma nova instância de `Node` é criada e adicionada no vetor de filhos daquele nó. Essa nova instância recebe como grade o estado do pai com a ação de preencher a posição vazia com o valor do iterador executada e a profundidade e custo iguais aos do pai incrementados de um. Após o *loop*, a lista de filhos é retornada.

4. Metodologia

A seguir, os algoritmos de busca implementados serão discutidos, juntamente das heurísticas adotadas para a busca informada e, por fim, o processo de experimentação: quais instâncias foram utilizadas e quais foram as principais métricas avaliadas.

4.1. Busca sem informação

Também conhecida como busca cega ou busca desinformada, é uma categoria de algoritmos de busca caracterizada pelas estratégias não possuírem informação adicional sobre os estados além daquela provida pela definição do problema [Russell and Norvig 2020].

Por esse motivo, são estratégias mais genéricas, que exploram o espaço de estados sistematicamente e, por isso, podem não ser tão eficientes. No entanto, a depender da forma como o problema é modelado e como a expansão dos estados ocorre, alguns problemas podem se beneficiar da abordagem de exploração do espaço de estados de alguns desses algoritmos.

De fato, a principal característica que distingue os algoritmos dessa categoria é a ordem com que os nós são expandidos e explorados. Em geral, cada um irá utilizar um tipo de *buffer* diferente (FIFO, LIFO, etc) para atingir a ordem desejada. As subseções a seguir detalham como os algoritmos *Breadth-first Search*, *Uniform-Cost Search* e *Iterative Deepening Search* funcionam, como os nós são explorados em cada abordagem, entre outras informações.

4.1.1. *Breadth-first Search*

A *Breadth-first Search* (BFS) é um algoritmo de busca que atravessa a árvore de estados expandindo todos os nós de uma mesma profundidade antes de passar para a próxima. Para atingir esse efeito, um *buffer* FIFO (fila) é utilizado. Na implementação realizada, a estrutura de fila implementada pela biblioteca padrão de C++ foi utilizada como *buffer* para garantir as características de exploração da BFS.

Inicialmente, o nó raiz é adicionado ao *buffer*. Em seguida, um *loop* é executado até que, em alguma iteração, o a fila esteja vazia (todos os estados tenham sido explorados). Em cada iteração, o topo da fila é removido e os filhos daquele nó são gerados. Para cada filho, o algoritmo verifica se o estado dele é um estado objetivo. Caso sim, esse estado é retornado. Caso não, ele insere o filho na fila, para que ele seja expandido no futuro. Se o algoritmo não retornar e a fila ficar vazia, um erro é retornado, informando que a BFS não foi capaz de encontrar uma solução.

A *Breadth-first Search* é uma estratégia bastante apropriada quando todas as ações realizadas sobre um estado possuem o mesmo custo. No entanto, os requisitos de tempo de execução e, principalmente, de memória são bastante altos, especialmente quando o estado objetivo é encontrado em profundidades maiores na árvore. Por fim, o algoritmo é completo, isto é, acha a solução se ela existir, e é ótimo, desde que o custo seja uma função não decrescente da profundidade do nó [Russell and Norvig 2020].

4.1.2. *Uniform-Cost Search*

A *Uniform-Cost Search* (UCS), ou também conhecida como algoritmo de Dijkstra, é uma estratégia de busca baseada na exploração dos nós que possuem o menor custo na iteração. É uma abordagem bastante semelhante a BFS mas que se sobressai em situações onde o custo das ações é diferente [Russell and Norvig 2020]. Caso os custos das ações sejam iguais, a UCS performa de forma similar a uma BFS.

A ideia aqui é utilizar uma fila de prioridade (*heap*) para armazenar os estados, de modo que eles sejam ordenados pelo custo associado. Assim, a cada iteração, o nó com o menor custo é retirado da *heap* e processado. O custo dos nós é calculado a partir da soma do custo do nó pai com o custo da ação que foi realizada para que o estado daquele nó filho fosse gerado.

Em C++, a implementação de fila de prioridade da biblioteca padrão é uma *max heap*. Portanto, para atingir o mesmo efeito de uma *min heap*, todos os algoritmos que utilizam a fila de prioridade como *buffer* inserem a função de custo associada ao nó com o valor invertido. Assim, a operação de recuperar o elemento do topo retorna o maior custo negativo que, se invertido de volta, se trata do menor custo positivo.

Dessa forma, na BFS, a árvore de estados é explorada em ondas uniformes com relação a profundidade dos nós e na UCS em ondas uniformes com relação ao custo do caminho de ações que está sendo construído. O algoritmo é completo e é ótimo com relação ao custo, isto é, a primeira solução que ele encontra tem o custo tão pequeno quanto o custo de qualquer nó que esteja na fronteira naquela iteração.

4.1.3. *Iterative Deepening Search*

A *Iterative Deepening Search* (IDS) é uma variação de outro algoritmo de busca: a *Depth-limited Search*. A ideia aqui é realizar a busca em profundidade (*Depth-first Search* ou DFS) em profundidades limitadas mas que crescem a cada iteração. Ou seja, na primeira iteração, $l = 0$ então apenas a raiz é explorada. Na segunda, $l = 1$ então a raiz e seus filhos são explorados. Isso ocorre sucessivamente até que um estado objetivo seja encontrado.

Esse algoritmo resolve tanto o problema do caminho infinito da DFS quanto o problema do limite insuficiente da *Depth-limited Search*. Dessa forma, a IDS combina os benefícios da DFS e da BFS: a complexidade de espaço é linear e o algoritmo é completo e ótimo para espaços de estados finitos e sem ciclos [Russell and Norvig 2020].

Apesar de parecer um algoritmo custoso, visto que os nós mais ao topo da árvore são explorados repetidas vezes, para muitos espaços de estados a maior parte dos nós está

em profundidades maiores da árvore. Portanto, não faz muita diferença, do ponto de vista de tempo de execução, que alguns nós sejam repetidos no processo.

Por todos esses motivos, a *Iterative Deepening Search* é a abordagem não informada preferida quando o espaço de estados não cabe na memória e a profundidade da solução buscada não é conhecida (pode ser arbitrariamente profunda) [Russell and Norvig 2020].

4.2. Busca com informação

Os algoritmos de busca informada, por sua vez, aproveitam de conhecimentos específicos que vão além da definição do problema para encontrar soluções de forma mais eficiente do que as estratégias desinformadas [Russell and Norvig 2020].

A ideia principal por traz dessa abordagem é aproveitar de heurísticas existentes para o problema para determinar quais nós parecem ser "mais promissores". Esses nós serão priorizados durante a exploração, com a intuição de que isso irá acelerar a descoberta do estado objetivo. A velocidade com que o nó objetivo será encontrado, no entanto, depende da qualidade da heurística.

As subseções a seguir detalham como os algoritmos *Greedy Best-first Search* e *A** funcionam, enfatizando como os nós são explorados e como as funções heurísticas ajudam no processo de busca. A subseção 4.2.3 aprofunda a discussão a respeito das heurísticas escolhidas em cada uma das implementações.

4.2.1. *Greedy Best-first Search*

A *Greedy Best-first Search* é uma estratégia de busca que prioriza a expansão de nós com o menor valor $h(n)$, isto é, o nó que aparenta estar mais próximo do objetivo segundo a nossa estimativa. Com isso, fazemos com que a função de avaliação seja $f(n) = h(n)$, esperando que isso faça com que a expansão leve a um estado objetivo mais rapidamente [Russell and Norvig 2020].

Nesse contexto, $h(n)$ se trata de uma heurística para o problema. Sendo assim, qualquer função aproximativa pode ser utilizada. No entanto, o processo de estabelecer heurísticas que fazem sentido e que se encaixam melhor no problema necessita de observações que vão além das definições do problema. Por esse motivo, funções heurísticas, em geral, se aproveitam de características e propriedades específicas do problema ou do domínio do mesmo para atingir resultados melhores.

O algoritmo é chamado de guloso justamente por escolher a cada iteração o nó que aparenta estar mais próximo do objetivo para ser expandido, segundo a heurística considerada. No entanto, é importante ter em mente que nem sempre a melhor escolha local leva ao melhor resultado global, resultados piores podem ser obtidos.

A *Greedy Best-first Search* é um algoritmo de busca completo em espaços de estados finitos mas não em espaços de estados infinitos. Ela não é ótima e pode ter uma complexidade de tempo e espaço alta, portanto, a escolha de heurísticas adequadas é fundamental para que o algoritmo seja eficiente e os piores casos para as complexidades de espaço e tempo não sejam atingidos [Russell and Norvig 2020].

4.2.2. *A* Search*

A *A* Search* é a estratégia de busca mais comum dentre os algoritmos de busca informada. Ela continua usando o mesmo conceito de função de avaliação que a *Greedy Best-first Search*, mas essa função passar a levar em conta também o custo do caminho até o nó atual. Ou seja, a função de avaliação pode ser dada por $f(n) = g(n) + h(n)$, onde $g(n)$ é o custo do caminho até o nó atual e $h(n)$ é o custo estimado do menor caminho do nó atual até o nó objetivo. Por esse motivo, a função de avaliação do *A** é vista como o custo estimado do melhor caminho que passa por n até o objetivo.

Do ponto de vista de implementação, o algoritmo é muito semelhante a *Uniform-Cost Search*, assim como a *Greedy Best-first Search* também era. A diferença principal acontece no cálculo do custo do nó, onde $f(n)$ é calculado como $g(n)+h(n)$ e não apenas $g(n)$, como acontece na UCS.

O algoritmo é completo e ótimo, desde que a heurística escolhida seja admissível, ou seja, que ela nunca superestime o custo para alcançar o nó objetivo. Caso a heurística escolhida não seja admissível, o algoritmo pode ou não ser ótimo em relação ao custo. Existem casos nos quais sabe-se que a solução encontrada é ótima, no entanto, não há garantia que ela sempre será, como acontece quando a heurística é admissível [Russell and Norvig 2020].

4.2.3. Heurísticas adotadas

Nas implementações realizadas para a *Greedy Best-first Search* e a *A* Search*, duas heurísticas diferentes foram utilizadas. Para o primeiro algoritmo, a heurística empregada leva em conta o número de possibilidades para cada célula vazia na grade. Para a *A* Search*, a heurística leva em conta o número de células que ainda precisam ser preenchidas para guiar a busca.

Na primeira heurística, a ideia é definir como função de avaliação para a Busca Gulosa a soma de quantos números podem ser adequadamente colocados em cada célula vazia. A intuição por traz é que se preferenciarmos estados com menos possibilidades de preenchimento, estaríamos expandindo os nós que possuem mais restrições associadas à grade e, portanto, a exploração da árvore sob essas condições deve levar mais rapidamente à solução ou a identificação de que aquele ramo não leva a um estado objetivo.

A segunda heurística leva em consideração a expansão de estados com o menor número de células vazias. A intuição dessa heurística é, de certa forma, semelhante a da primeira: queremos preferenciar a expansão de nós cujos estados tenham a maior quantidade de células preenchidas, seguindo a ideia que isso faria com que a exploração chegasse mais rapidamente a um estado objetivo ou que a detecção de ramos que não fossem levar a uma solução fosse feita prematuramente. Assim, a função de avaliação no *A* Search* será a soma do custo do nó com o número de células vazias no estado.

4.3. Experimentos

Os experimentos e procedimentos realizados utilizaram como entrada um conjunto de grades incompletas do Sudoku[1]. Todas as instâncias possuem solução e são divididas da seguinte forma:

- 9 instâncias fáceis;
- 10 instâncias médias;
- 13 instâncias difíceis.

Apesar da distinção entre a dificuldade das instâncias não se dar com base no número de células inicialmente preenchidas, uma análise que leva em conta esse número e as métricas obtidas para cada execução será realizada para avaliar se a dificuldade pode ou não ser medida exclusivamente dessa forma.

Para a execução dos testes, foi desenvolvido um *bash script* que compila o código e executa cada algoritmo contra cada uma das instâncias definidas. Após cada execução, o *script* salva os resultados obtidos em um arquivo CSV, para que a análise exploratória possa ser realizada e o desempenho de cada algoritmo possa ser devidamente avaliado.

Para avaliar o desempenho dos algoritmos, os principais critérios analisados foram o número de estados expandidos até a solução ser encontrada e o tempo de execução de cada algoritmo para cada entrada. Todos os testes foram executados em um computador com processador Intel i5-10210U (8) @ 4.200GHz e 20 GB de RAM.

5. Resultados

Ao executar os algoritmos contra as instâncias, vemos nas tabelas 1 e 2, para cada dificuldade de instância, a média de estados expandidos e o temo médio despendido computando cada solução, respectivamente.

Dificuldade	BFS	UCS	IDS	*Greedy*	*A**
Fácil	892.77	895.11	23005.89	191.55	197.22
Média	2028.60	2029.60	55702.40	1300.30	548.00
Difícil	30401.85	30402.85	889923.31	5982.46	11785.23

Tabela 1. Número de estados expandidos

Dificuldade	BFS	UCS	IDS	*Greedy*	*A**
Fácil	10.22	12.44	599.33	6.55	3.00
Média	26.60	27.90	1440.50	43.10	9.00
Difícil	622.15	435.69	22483.08	193.08	168.31

Tabela 2. Tempo médio de execução, em milissegundos

Analisando a tabela 1, é possível perceber uma semelhança entre o número de estados expandidos pela BFS e pela UCS. Como no Sudoku a única ação existente é a de preencher uma posição da grade e cada preenchimento tem custo um, a busca não é beneficiada pela ordenação induzida pela fila de prioridade (menor caminho). O mesmo

[1]É possível realizar o *download* das instâncias nesse link.

cenário pode ser observado para a média dos tempos de execução listados em 2, salvo apenas de uma pequena variação para as instâncias difíceis. Portanto, graças as características do problema e de sua modelagem, a UCS se degenera em uma BFS.

A IDS, como era de se esperar, expandiu muito mais estados e levou muito mais tempo que os outros algoritmos. Isso acontece porque o algoritmo promove a exploração repetida dos nós presentes nas profundidades mais rasas da árvore. Essa repetição provoca um aumento considerável no número de estados expandidos, especialmente para as dificuldades média e difícil, e, por conseguinte, um aumento considerável no tempo de execução.

Os algoritmos de busca informada apresentaram uma média de expansão de estados não padronizada. Para as instâncias de dificuldade média, o $A*$ expandiu, em média, apenas metade dos nós que a Busca Gulosa expandiu. Em contrapartida, essa situação se inverte quando analisamos as instâncias difíceis, onde o $A*$ expandiu o dobro de nós que a Busca Gulosa. Isso pode estar relacionado com as qualidades das heurísticas empregadas.

Com relação ao tempo computacional, a variação foi pequena mas o $A*$ obteve uma média de tempo melhor que a Busca Gulosa para todas as dificuldades de instâncias testadas. Esse resultado junto das médias obtidas para o número de estados expandidos nos leva a crer que a heurística usada pela Busca Gulosa parece ser um pouco melhor que a heurística do $A*$ do ponto de vista do número de estados explorados, mas é um pouco mais cara computacionalmente.

Por fim, é possível perceber que os algoritmos de busca informada se saíram melhor que os algoritmos de busca cega para todas as dificuldades em ambas as métricas. Assim, é possível afirmar que as heurísticas foram capazes de guiar a busca para que menos estados precisassem ser expandidos e, por conseguinte, menos tempo computacional fosse despendido com cada instância, até que a solução fosse encontrada.

É importante ressaltar que esse foi o caso por causa do problema e das heurísticas adotadas. Se heurísticas com custo computacional alto fossem utilizadas, pode ser que o número de estados expandidos ainda fosse menor mas o tempo de execução desses algoritmos fosse maior que os de busca não informada. Portanto, só é possível afirmar isso por causa da análise dos resultados obtidos, da modelagem feita para o problema e do custo computacional das heurísticas escolhidas.

Para avaliar a proposição de que a heurística da Busca Gulosa é melhor que a heurística do $A*$, os gráficos 1 e 2 foram gerados. O primeiro compara o número de estados expandidos e o segundo o tempo computacional para cada instância no conjunto de experimentos.

É possível perceber que existem duas instâncias problemáticas. Essas instâncias demandam uma exploração mais profunda da árvore de estados e, por isso, um tempo computacional maior. Esses *outliers*, em especial o que possui o maior pico, foram os responsáveis pela variação não regular nas médias observadas na tabela 1.

Para a instância que apresentou o maior pico, o número de estados expandidos pelo $A*$ é três vezes maior que o número de estados expandidos pela Busca Gulosa. No entanto, a diferença de tempo computacional entre os algoritmos não é tão significativa assim e essa situação se repete, dadas as devidas escalas, para as outras instâncias também.

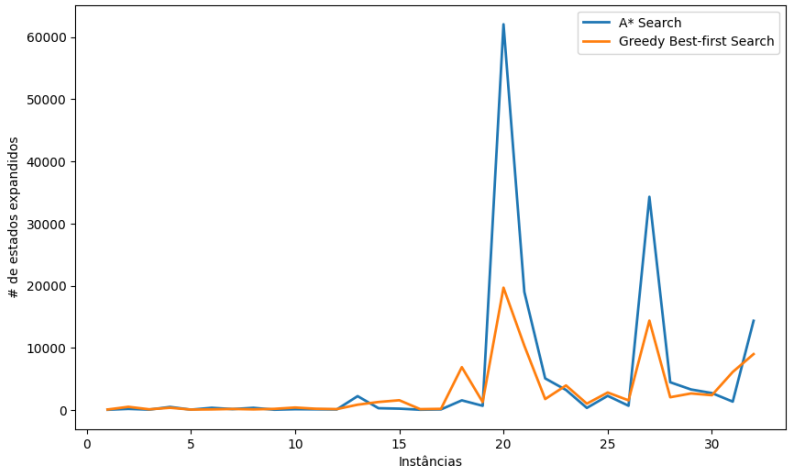

Figura 1. Gráfico de linha com a comparação do número de estados expandidos por cada algoritmo de busca informada

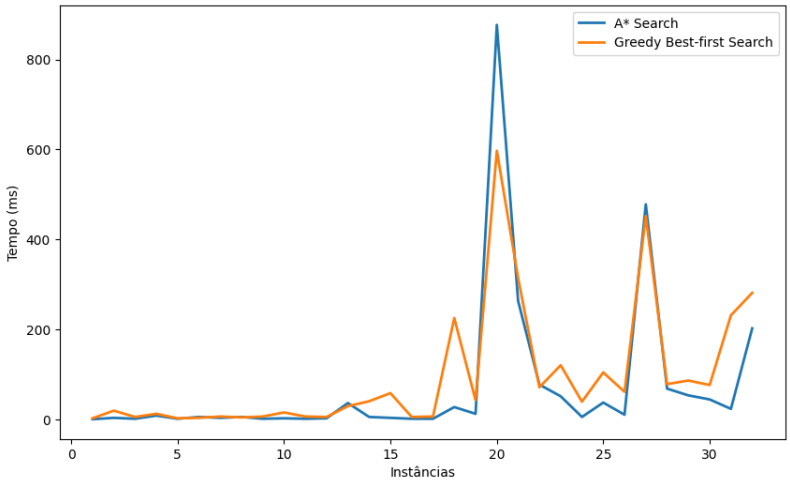

Figura 2. Gráfico de linha com a comparação do tempo computacional gasto por cada algoritmo de busca informada

Isso pode evidenciar, de fato, que a heurística da Busca Gulosa explora a árvore de estados de maneira mais eficiente, o que leva a um número menor de estados explorados, mas tem um custo computacional maior.

Por fim, os gráficos 3 e 4 foram gerados para comparar as métricas avaliadas para cada algoritmo com relação ao número de dicas, ou seja, o número de células inicialmente preenchidas em cada instância. Os estados expandidos e os tempos da IDS foram omitidos para facilitar a visualização dos gráficos mas, fora a variação de escala, o comportamento apresentado por ele acompanhou o dos outros algoritmos.

Assim como constatado anteriormente, o número de células preenchidas inicialmente parece ter um impacto direto na dificuldade das instâncias. As instâncias com poucas dicas apresentam os maiores tempos e número de estados expandidos enquanto o comportamento contrário acontece para as instâncias com muitas dias.

No entanto, é possível perceber flutuações, especialmente para as instâncias mé-

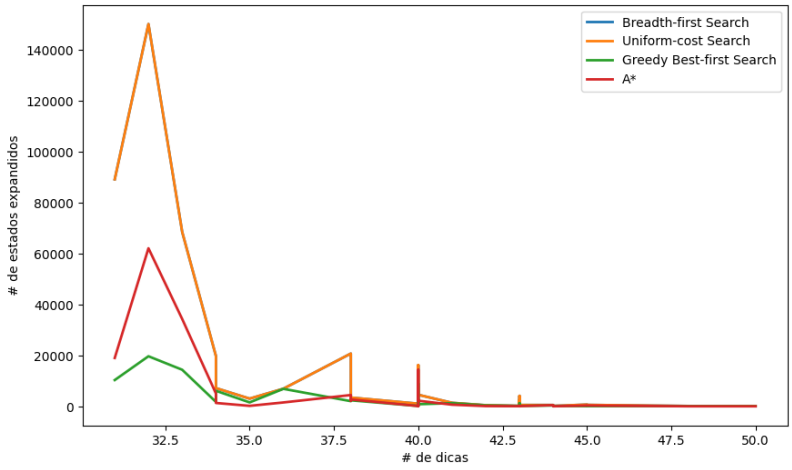

Figura 3. Gráfico de linha com a comparação do número de dicas em cada instância pelo número de estados expandidos por cada algoritmo

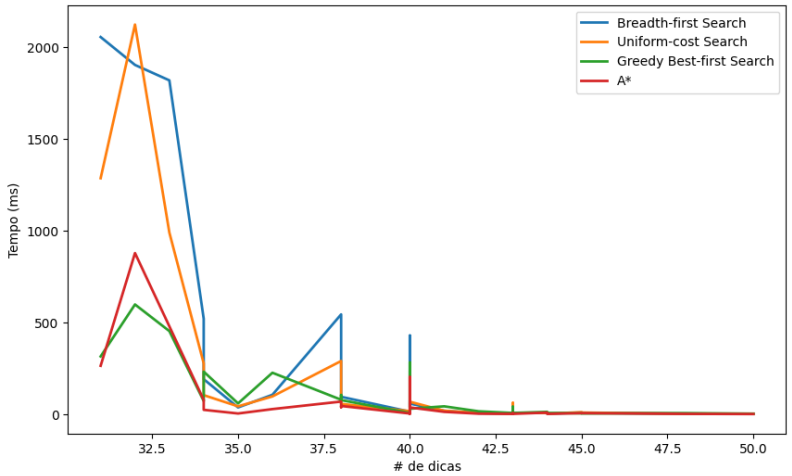

Figura 4. Gráfico de linha com a comparação do número de dicas em cada instância pelo tempo computacional gasto por cada algoritmo

dias e difíceis. Isso pode indicar que, apesar do número de dicas ter uma grande influência na dificuldade, outros fatores, como as posições de cada célula preenchida, também parecem ter algum impacto na dificuldade da instância.

6. Conclusão

Com o desenvolvimento do programa, foi possível compreender as diferenças de implementação e eficiência para os cinco algoritmos de busca implementados. O processo de experimentação reforçou as qualidades de cada algoritmo mas, sobretudo, os problemas, especialmente aqueles atrelados à modelagem computacional do Sudoku.

De modo geral, foi constatado que a *Uniform-Cost Search* não performa bem para esse problema, uma vez que as ações tem custo um e, portanto, a busca não é beneficiada ao preferenciar a expansão de nós que minimizem o caminho construído. Além disso, a *Iterative Deepening Search* não performou bem com relação as métricas avaliadas e aos outros algoritmos.

Por fim, foi possível perceber os ganhos de desempenho obtidos ao utilizar estratégias de busca informada em detrimento as estratégias de busca cega. Mais que isso, foi possível observar, também, algumas diferenças nas métricas que foram causadas pela heurística escolhida. Apesar do impacto não ter sido significativo nesse caso, para casos onde a heurística é mais cara computacionalmente, o *trade-off* dessa complexidade em relação ao desempenho obtido com ela deve ser muito bem avaliado.

Referências

[Russell and Norvig 2020] Russell, S. and Norvig, P. (2020). *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson.

[Wikipedia 2024] Wikipedia (2024). Sudoku. `https://en.wikipedia.org/wiki/Sudoku`. Acesso em: 03 de mai. de 2024.

[Yato and Seta 2003] Yato, T. and Seta, T. (2003). Complexity and completeness of finding another solution and its application to puzzles. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 86-A(5):1052–1060.